> home > about > feedback > login

US Patent & Trademark Office

Try the *new* Portal design

Give us your opinion after using it.

Search Results

Search Results for: **[matlab and shap* and infer* <AND>(meta_published_date <= 01-01-2001)]**
Found **19** of **56 searched out of 126,861.**

Search within Results

> Advanced Search

> Search Help/Tips

Sort by: Title Publication Publication Date **Score** Binder

Results 1 - 19 of 19 short listing

1 A case for source-level transformations in MATLAB 100%
Vijay Menon , Keshav Pingali
ACM SIGPLAN Notices , Proceedings of the 2nd conference on Domain-specific languages December 1999
Volume 35 Issue 1

In this paper, we discuss various performance overheads in MATLAB codes and propose different program transformation strategies to overcome them. In particular, we demonstrate that high-level source-to-source transformations of MATLAB programs are effective in obtaining substantial performance gains regardless of whether programs are interpreted or later compiled into C or FORTRAN. We argue that automating such transformations provides a promising area of future research.

2 Techniques for the translation of MATLAB programs into Fortran 90 100%
Luiz De Rose , David Padua
ACM Transactions on Programming Languages and Systems (TOPLAS) March 1999
Volume 21 Issue 2

This article describes the main techiques developed for FALCON's MATLAB-to-Fortran 90 compiler. FALCON is a programming environment for the development of high-performance scientific programs. It combines static and dynamic inference methods to translate MATLAB programs into Fortran 90. The static inference is supported with advanced value propagation techniques and symbolic algorithms for subscript analysis. Experiments show that FALCON's MATLAB translator can generate code that performs m ...

3 A MATLAB to Fortran 90 translator and its effectiveness 100%
Luiz De Rose , David Padua
Proceedings of the 10th international conference on Supercomputing January 1996

4 High-level semantic optimization of numerical codes 100%
Vijay Menon , Keshav Pingali
Proceedings of the 13th international conference on Supercomputing May 1999

5 Correctly detecting intrinsic type errors in typeless languages such as MATLAB 100%
Pramod G. Joisha , Prithviraj Banerjee
ACM SIGAPL APL Quote Quad , Proceedings of the 2001 conference on APL: an arrays odyssey December 2000
Volume 31 Issue 2

Among the main impediments that languages such as MATLAB and APL present to a compiler is the lack of an explicit declaration for a variable's type, The determination of this important attribute could allow a compiler to generate more efficient code, and is a problem that has been extensively studied in the past. This paper revisits this problem but unlike prior efforts, the objective is a uniform approach to type estimation that also accommodates type incorrect programs in a way that facilitate ...

6 Expokit: a software package for computing matrix exponentials 89%
Roger B. Sidje
ACM Transactions on Mathematical Software (TOMS) March 1998
Volume 24 Issue 1

Expokit provides a set of routines aimed at computing matrix exponentials. More precisely, it computes either a small matrix exponential in full, the action of a large sparse matrix exponential on an operand vector, or the solution of a system of linear OBEs with constant inhomogeneity. The backbone of the sparse routines consists of matrix-free Krylov subspace projection methods (Arnoldi and Lanczos processes), and that is why the toolkit is capable of coping with sparse matrices of large ...

7 Efficient support of parallel sparse computation for array intrinsic functions of Fortran 90 88%
Rong-Guey Chang , Tyng-Ruey Chuang , Jenq Kuen Lee
Proceedings of the 12th international conference on Supercomputing July 1998

8 Query processing techniques for arrays 85%
Arunprasad P. Marathe , Kenneth Salem
ACM SIGMOD Record , Proceedings of the 1999 ACM SIGMOD international conference on Management of data June 1999
Volume 28 Issue 2

Arrays are an appropriate data model for images, gridded output from computational models, and other types of data. This paper describes an approach to array query processing. Queries are expressed in AML, a logical algebra that is easily extended with user-defined functions to support a wide variety of array operations. For example, compression, filtering, and algebraic operations on images can be described. We show how AML expressions involving such operations can be treated declaratively ...

9 Papers: Suitability assessment of software developers: a fuzzy approach 82%
Anu Singh Lather , Shakti Kumar , Yogesh Singh
ACM SIGSOFT Software Engineering Notes May 2000

Volume 25 Issue 3

The right selection of software personnels helps keep project cost low, deliver better quality software and avoids the schedule slippage of a software project. We have identified the 3 most essential abilities/aptitudes which can decide comparative merit of the software developers. These are Verbal Reasoning (VR), Numerical Ability (NA) and Abstract Reasoning (AR). As a fuzzy model is a best choice for managing ambiguous, doubtful, contradicting and diverging opinions we propose a three input an ...

10 ftd: an exact frequency to time domain conversion for reduced order RLC interconnect models 82%

Ying Liu , Lawrence T. Pileggi , Andrzej J. Strojwas

Proceedings of the 35th annual conference on Design automation conference

May 1998

Recursive convolution provides an exact solution for interfacing reduced-order frequency domain representations with discrete time domain models of piecewise linear voltage waveforms. The state-space method is more efficient, but not exact, and can sometimes produce large time domain errors. This paper presents a new algorithm, ftd (frequency to time domain), for incorporating linear frequency domain macro-models into time domain simulators. ftd provides accuracy equivalent to recur ...

11 Sufficiency analysis for the calculus of variations 80%

Robert M. Corless

Proceedings of the international symposium on Symbolic and algebraic computation August 1994

12 Reports from related meetings: Interface '99: a data mining overview 77%

Arnold Goodman

ACM SIGKDD Explorations Newsletter January 2000

Volume 1 Issue 2

This personal overview of Interface '99 is intended to communicate its meaning and relevance to SIGKDD, as well as provide valuable information on trends within the Interface for data miners seeking to learn more about statistics. In addition, it is the newest link in a bridge between the Interface and KDD begun by References 2-4 and the sessions on KDD at Interface '98 and Interface '99.

13 GAMS: a framework for the management of scientific software 77%

Ronald F. Boisvert , Sally E. Howe , David K. Kahaner

ACM Transactions on Mathematical Software (TOMS) March 1986

Volume 11 Issue 4

The Guide to Available Mathematical Software (GAMS) provides a framework for both a scientist-end-user and a librarian-maintainer to deal with large quantities of mathematical and statistical software. This framework includes a classification scheme for mathematical and statistical software, a database system to manage information about this software, and both an on-line interactive consulting system and a printed catalog for providing users with access to this information. A description is ...

14 FORTRAN FUTURES: Abstracts 77%

ACM SIGPLAN Fortran Forum December 1998

Volume 17 Issue 3

15 Style machines 77%
Matthew Brand , Aaron Hertzmann
Proceedings of the 27th annual conference on Computer graphics and interactive techniques July 2000
We approach the problem of stylistic motion synthesis by learning motion patterns from a highly varied set of motion capture sequences. Each sequence may have a distinct choreography, performed in a distinct sytle. Learning identifies common choreographic elements across sequences, the different styles in which each element is performed, and a small number of stylistic degrees of freedom which span the many variations in the dataset. The learned model can synthesize novel motion data in any ...

16 Design of an adaptive control system for DC servo motor 77%
F. Remy , M. Weck
Proceedings of the 1995 ACM symposium on Applied computing February 1995

17 Towards automated synthesis of data mining programs 77%
Wray Buntine , Bernd Fischer , Thomas Pressburger
Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining August 1999

18 PELLPACK: a problem-solving environment for PDE-based applications on multicomputer platforms 77%
E. N. Houstis , J. R. Rice , S. Weerawarana , A. C. Catlin , P. Papachiou , K.-Y. Wang , M. Gaitatzes
ACM Transactions on Mathematical Software (TOMS) March 1998
Volume 24 Issue 1
The article presents the software architecture and implementation of the problem-solving environment (PSE) PELLPACK for modeling physical objects described by partial differential equations (PDEs). The scope of this PSE is broad, as PELLPACK incorporates many PDE solving systems, and some of these, in turn, include several specific PDE solving methods. Its coverage for 1D, 2D. and 3D elliptic or parabolic problems is quite broad, and it handles some hyperbolic problems, Since a PSE should p ...

19 Modeling the benefits of mixed data and task parallelism 77%
Soumen Chakrabarti , James Demmel , Katherine Yelick
Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures July 1995

Results 1 - 19 of 19 short listing

IEEE HOME | SEARCH IEEE | SHOP | WEB ACCOUNT | CONTACT IEEE

IEEE

Membership Publications/Services Standards Conferences Careers/Jobs

IEEE Xplore RELEASE 1.6

Welcome
United States Patent and Trademark Office

Help FAQ Terms IEEE Peer Review Quick Links » Se

Welcome to IEEE Xplore
- Home
- What Can I Access?
- Log-out

Tables of Contents
- Journals & Magazines
- Conference Proceedings
- Standards

Search
- By Author
- Basic
- Advanced

Member Services
- Join IEEE
- Establish IEEE Web Account
- Access the IEEE Member Digital Library

Your search matched **1** of **1000582** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevance** **Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or enter new one in the text box.

matlab and shap* and infer* Search

☐ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

1 **Match virtual machine: an adaptive runtime system to execute MAT in parallel**
Haldar, M.; Nayak, A.; Kanhere, A.; Joisha, P.; Shenoy, N.; Choudhary, A.; Banerjee, P.;
Parallel Processing, 2000. Proceedings. 2000 International Conference on , 21 Aug. 2000
Pages:145 - 152

[Abstract] [PDF Full-Text (644 KB)] **IEEE CNF**

Home | Log-out | Journals | Conference Proceedings | Standards | Search by Author | Basic Search | Advanced Search | Join IEEE | Web Account | New this week | OPAC Linking Information | Your Feedback | Technical Support | Email Alerting | No Robots Please | Release Notes | IEEE Online Publications | Help | FAQ| Terms | Back to Top

NPL Virtual Library

My List - 0 Help

Search

Main Search | Advanced Keyword Search | Search History

Search: Author Keyword Prithviraj Refine Search

You're searching: **Scientific and Technical Information Center**

Sort by: Publication date
Limit by: Select...

Search Results

2 titles matched: Prithviraj

1. **Proceedings of the 1995 International Conference on Parallel Processing, August 14-18, 1995 / sponsored by The Pennsylvania State University.**
   by *International Conference on Parallel Processing (24th : 1995 : Oconomowoc, Wis.)*
   CRC Press, c1995.
   Call No.: QA76.58 .I55 1995
   Add to my list

2. **Proceedings of the 1995 International Conference on Parallel Processing, August 14-18, 1995 / sponsored by The Pennsylvania State University.**
   by *International Conference on Parallel Processing (24th : 1995 : Oconomowoc, Wis.)*
   CRC Press, c1995.
   Call No.: QA76.58 .I55 1995
   Add to my list

Add/Remove MyList (max=100, ie. 1,2 5-20) 1-2 Add Remove

Mastering MATLAB 5 : a comprehensive tutorial and reference /

Author:	Hanselman, Duane C. Littlefield, Bruce.
Imprint:	Upper Saddle River, N.J. : Prentice Hall, c1998.
Contents:	1. Getting Started -- 2. Basic Features -- 3. The Command Window -- 4. Script M-Files -- 5. File and Directory Management -- 6. Arrays and Array Operations -- 7. Multidimensional Arrays -- 8. Relational and Logical Operations -- 9. Set, Bit, and Base Functions -- 10. Character Strings -- 11. Time Functions -- 12. Cell Arrays and Structures -- 13. Control Flow -- 14. Function M-Files -- 15. Debugging and Profiling Tools -- 16. Numerical Linear Algebra -- 17. Data Analysis -- 18. Polynomials -- 19. Interpolation -- 20. Cubic Splines -- 21. Fourier Analysis -- 22. Optimization -- 23. Integration and Differentiation -- 24. Ordinary Differential Equations -- 25. Object-Oriented Programming -- 26. 2-D Graphics -- 27. 3-D Graphics -- 28. Using Color and Light -- 29. Images, Movies, and Sound -- 30. Printing and Exporting Graphics -- 31. Handle Graphics -- 32. Graphical User Interfaces -- 33. Dialog Boxes and Requesters -- 34. Help -- 35. Internet Resources -- 36. The Mastering MATLAB Toolbox -- App. A. MATLAB Function Listing -- App. B. Axes Object Properties -- App. C. Figure Object Properties -- App. D. Image Object Properties -- App. E. Light Object Properties -- App. F. Line Object Properties -- App. G. Patch Object Properties -- App. H. Root Object Properties -- App. I. Surface Object Properties -- App. J. Text Object Properties -- App. K. UIControl Object Properties -- App. L. UIMenu Object Properties.
Notes:	Includes bibliographical references (p. 521-526) and index.
ISBN:	0138583668
Subjects:	MATLAB. Numerical analysis -- Data processing
Series:	The MATLAB curriculum series
Description:	xviii, 638 p. : ill. ; 24 cm.

dblp.uni-trier.de

Prithviraj Banerjee

List of publications from the DBLP Bibliography Server - FAQ

Ask others: ACM DL - ACM Guide - CiteSeer - CSB - Google

Home Page

		2003
186	EE	Pramod G. Joisha, Prithviraj Banerjee: The MAGICA Type Inference Engine for MATLAB. CC 2003: 121-125
185	EE	Prithviraj Banerjee, Debabrata Bagchi, Malay Haldar, Anshuman Nayak, Victor Kim, R. Uribe: Automatic Conversion of Floating Point MATLAB Programs into Fixed Point FPGA Based Hardware Design. FCCM 2003: 263-264
184	EE	Alex Jones, Prithviraj Banerjee: An Automated and Power-Aware Framework for Utilization of IP Cores in Hardware Generated from C Descriptions Targeting FPGAs. FCCM 2003: 284-285
183	EE	Prithviraj Banerjee, V. Saxena, J. Uribe, Malay Haldar, Anshuman Nayak, Victor Kim, Debabrata Bagchi, Satrajit Pal, Nikhil Tripathi, R. Anderson: Making area-performance tradeoffs at the high level using the AccelFPGA compiler for FPGAs. FPGA 2003: 237
182	EE	Alex Jones, Prithviraj Banerjee: An automated and power-aware framework for utilization of IP cores in hardware generated from C descriptions targeting FPGAs. FPGA 2003: 244
181	EE	Pramod G. Joisha, Prithviraj Banerjee: Static array storage optimization in MATLAB. PLDI 2003: 258-268
180	EE	Amitabh Mishra, Prithviraj Banerjee: An Algorithm-Based Error Detection Scheme for the Multigrid Method. IEEE Trans. Computers 52(9): 1089-1099 (2003)
179	EE	Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: Reducing False Sharing and Improving Spatial Locality in a Unified Compilation Framework. IEEE Trans. Parallel Distrib. Syst. 14(4): 337-354 (2003)
		2002
178	EE	Anshuman Nayak, Malay Haldar, Alok N. Choudhary, Prithviraj Banerjee: Accurate Area and Delay Estimators for FPGAs. DATE 2002: 862-869
177	EE	Prithviraj Banerjee, Malay Haldar, Anshuman Nayak, Victor Kim, Debabrata Bagchi, Satrajit Pal, Nikhil Tripathi: A Behavioral Synthesis Tool for Exploiting Fine Grain Parallelism in FPGAs. IWDC 2002: 246-256
176	EE	Venkatram Krishnaswamy, Gagan Hasteer, Prithviraj Banerjee: Automatic Parallelization of Compiled Event Driven VHDL Simulation. IEEE Trans. Computers 51(4): 380-394 (2002)
		2001
175	EE	Pramod G. Joisha, Prithviraj Banerjee: Correctly detecting intrinsic type errors in typeless languages such as MATLAB. APL 2001: 7-21

174	EE	Daniel J. Palermo, Eugene W. Hodges IV, Prithviraj Banerjee: Compiler Optimization of Dynamic Data Distributions for Distributed-Memory Multicomputers. Compiler Optimizations for Scalable Parallel Systems Languages 2001: 445-484
173	EE	Anshuman Nayak, Malay Haldar, Alok N. Choudhary, Prithviraj Banerjee: Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. DATE 2001: 722-728
172	EE	Malay Haldar, Anshuman Nayak, Alok N. Choudhary, Prithviraj Banerjee: A System for Synthesizing Optimized FPGA Hardware from MATLAB. ICCAD 2001: 314-319
171	EE	Dhruva R. Chakrabarti, Prithviraj Banerjee: Global optimization techniques for automatic parallelization of hybrid applications. ICS 2001: 166-180
170	EE	Pramod G. Joisha, Nagaraj Shenoy, Prithviraj Banerjee: Computing Array Shapes in MATLAB. LCPC 2001: 395-410
169	EE	Nagaraj Shenoy, Alok N. Choudhary, Prithviraj Banerjee: An algorithm for synthesis of large time-constrained heterogeneous adaptive systems. ACM Trans. Design Autom. Electr. Syst. 6 (2): 207-225 (2001)
168	EE	Mahmut T. Kandemir, J. Ramanujam, Alok N. Choudhary, Prithviraj Banerjee: A Layout-Conscious Iteration Space Transformation Technique. IEEE Trans. Computers 50(12): 1321-1336 (2001)
167	EE	Pramod G. Joisha, Prithviraj Banerjee: The Efficient Computation of Ownership Sets in HPF. IEEE Trans. Parallel Distrib. Syst. 12(8): 769-788 (2001)
166	EE	Mahmut T. Kandemir, Prithviraj Banerjee, Alok N. Choudhary, J. Ramanujam, Eduard Ayguadé: Static and Dynamic Locality Optimizations Using Integer Linear Programming. IEEE Trans. Parallel Distrib. Syst. 12(9): 922-941 (2001)
165		Dhruva R. Chakrabarti, Prithviraj Banerjee: Static Single Assignment Form for Message-Passing Programs. International Journal of Parallel Programming 29(2): 139-184 (2001)
164	EE	Yanhong Yuan, Prithviraj Banerjee: A Parallel Implementation of a Fast Multipole-Based 3-D Capacitance Extraction Program on Distributed Memory Multicomputers. J. Parallel Distrib. Comput. 61(12): 1751-1774 (2001)
2000		
163	EE	Malay Haldar, Anshuman Nayak, Alok N. Choudhary, Prithviraj Banerjee: Scheduling algorithms for automated synthesis of pipelined designs on FPGAs for applications described in MATLAB. CASES 2000: 85-93
162	EE	U. Nagaraj Shenoy, Prithviraj Banerjee, Alok N. Choudhary: A System-Level Synthesis Algorithm with Guaranteed Solution Quality. DATE 2000: 417-
161	EE	Zhi Alex Ye, Nagaraj Shenoy, Prithviraj Banerjee: A C compiler for a processor with a reconfigurable functional unit. FPGA 2000: 95-100
160	EE	Yanhong Yuan, Prithviraj Banerjee: Comparative Study of Parallel Algorithms for 3-D Capacitance Extraction on Distributed Memory Multiprocessors. ICCD 2000: 133-
159	EE	Malay Haldar, Anshuman Nayak, Abhay Kanhere, Pramod G. Joisha, Nagaraj Shenoy, Alok N. Choudhary, Prithviraj Banerjee: Match Virtual Machine: An Adaptive Runtime System to Execute MATLAB in Parallel. ICPP 2000: 145-152
158	EE	Victor Kim, Prithviraj Banerjee, Kaushik De: Fine-Grained Parallel VLSI Synthesis for Commercial CAD on a Network of Workstations. ICPP 2000: 421-
		Yanhong Yuan, Prithviraj Banerjee: A Parallel Implementation of a Fast Multipole Based 3-D

157	EE	Capacitance Extraction Program on Distributed Memory Multicomputer. IPDPS 2000: 323-330
156	EE	Zhi Alex Ye, Andreas Moshovos, Scott Hauck, Prithviraj Banerjee: CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. ISCA 2000: 225-235
155	EE	Pramod G. Joisha, Prithviraj Banerjee: Exploiting Ownership Sets in HPF. LCPC 2000: 259-273
154	EE	Mahmut T. Kandemir, Alok N. Choudhary, Prithviraj Banerjee, J. Ramanujam, Nagaraj Shenoy: Minimizing Data and Synchronization Costs in One-Way Communication. IEEE Trans. Parallel Distrib. Syst. 11(12): 1232-1251 (2000)
153	EE	Antonio Lain, Dhruva R. Chakrabarti, Prithviraj Banerjee: Compiler and Run-Time Support for Exploiting Regularity within Irregular Applications. IEEE Trans. Parallel Distrib. Syst. 11 (2): 119-135 (2000)
		1999
152		Prithviraj Banerjee, Viktor K. Prasanna, Bhabani P. Sinha: High Performance Computing - HiPC'99, 6th International Conference, Calcutta, India, December 17-20, 1999, Proceedings. Springer 1999
151	EE	Sumit Roy, Krishna P. Belkhale, Prithviraj Banerjee: An Approxmimate Algorithm for Delay-Constraint Technology Mapping. DAC 1999: 367-372
150	EE	Amitabh Mishra, Prithviraj Banerjee: An Algorithm Based Error Detection Scheme for the Multigrid Algorithm. FTCS 1999: 12-19
149		Yanhong Yuan, Prithviraj Banerjee: A Parallel 3-D Capacitance Extraction Program. HiPC 1999: 202-206
148	EE	Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: A Framework for Interprocedural Locality Optimization Using Both Loop and Data Layout Transformations. ICPP 1999: 95-102
147	EE	Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: On Reducing False Sharing while Improving Locality on Shared Memory Multiprocessors. IEEE PACT 1999: 203-211
146	EE	Dhruva R. Chakrabarti, Prithviraj Banerjee: A Novel Compilation Framework for Supporting Semi-Regular Distributions in Hybrid Applications. IPPS/SPDP 1999: 597-602
145	EE	Pramod G. Joisha, Prithviraj Banerjee: PARADIGM (version 2.0): A New HPF Compilation System. IPPS/SPDP 1999: 609-615
144	EE	Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: A Graph Based Framework to Detect Optimal Memory Layouts for Improving Data Locality. IPPS/SPDP 1999: 738-743
143	EE	Yanhong Yuan, Prithviraj Banerjee: Incremental capacitance extraction and its application to iterative timing-driven detailed routing. ISPD 1999: 42-47
142	EE	Mahmut T. Kandemir, Prithviraj Banerjee, Alok N. Choudhary, J. Ramanujam, Eduard Ayguadé: An integer linear programming approach for optimizing cache locality. International Conference on Supercomputing 1999: 500-509
141	EE	Dhruva R. Chakrabarti, Prithviraj Banerjee: Accurate Data and Context Management in Message-Passing Programs. LCPC 1999: 117-132
		Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: Improving

140		Locality Using a Graph-Based Technique for Detecting Memory Layouts of Arrays. PPSC 1999
139	EE	Mahmut T. Kandemir, Prithviraj Banerjee, Alok N. Choudhary, J. Ramanujam, Nagaraj Shenoy: A global communication optimization technique based on data-flow analysis and linear algebra. ACM Trans. Program. Lang. Syst. 21(6): 1251-1297 (1999)
138		Pradeep Prabhakaran, Prithviraj Banerjee: Parallel Algorithms for Force Directed Scheduling of Flattened and Hierarchical Signal Flow Graphs. IEEE Trans. Computers 48(7): 762-768 (1999)
137	EE	Mahmut T. Kandemir, Alok N. Choudhary, Nagaraj Shenoy, Prithviraj Banerjee, J. Ramanujam: A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts. IEEE Trans. Parallel Distrib. Syst. 10(2): 115-135 (1999)
136		John A. Chandy, Prithviraj Banerjee: A Parallel Circuit-Partitioned Algorithm for Timing-Driven Standard Cell Placement. J. Parallel Distrib. Comput. 57(1): 64-90 (1999)
135		Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: A Matrix-Based Approach to Global Locality Optimization. J. Parallel Distrib. Comput. 58(2): 190-235 (1999)
1998		
134	EE	Maogang Wang, Prithviraj Banerjee, Majid Sarrafzadeh: Potential-NRG: Placement with Incomplete Data. DAC 1998: 279-282
133	EE	Gagan Hasteer, Anmol Mathur, Prithviraj Banerjee: An Implicit Algorithm for Finding Steady States and its Application to FSM Verification. DAC 1998: 611-614
132	EE	Victor Kim, Prithviraj Banerjee: Parallel Algorithms for Power Estimation. DAC 1998: 672-677
131	EE	Sumit Roy, Harm Arts, Prithviraj Banerjee: PowerShake: A Low Power Driven Clustering and Factoring Methodology for Boolean Expressions. DATE 1998: 967-968
130	EE	Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Nagaraj Shenoy, Prithviraj Banerjee: Enhancing Spatial Locality via Data Layout Optimizations. Euro-Par 1998: 422-434
129	EE	Gagan Hasteer, Anmol Mathur, Prithviraj Banerjee: Efficient equivalence checking of multi-phase designs using retiming. ICCAD 1998: 557-562
128	EE	Sumit Roy, Harm Arts, Prithviraj Banerjee: PowerDrive: a fast, canonical POWER estimator for DRIVing synthEsis. ICCAD 1998: 601-606
127	EE	Mahmut T. Kandemir, Nagaraj Shenoy, Prithviraj Banerjee, J. Ramanujam, Alok N. Choudhary: Minimizing Data and Synchronization Costs in One-Way Communication. ICPP 1998: 180-188
126	EE	Zhaoyun Xing, Prithviraj Banerjee: A Parallel Algorithm for Timing-driven Global Routing for Standard Cells. ICPP 1998: 54-61
125	EE	Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: A Matrix-Based Approach to the Global Locality Optimization Problem. IEEE PACT 1998: 306-313
124	EE	Mahmut T. Kandemir, Prithviraj Banerjee, Alok N. Choudhary, J. Ramanujam, Nagaraj Shenoy: A Generalized Framework for Global Communication Optimization. IPPS/SPDP 1998: 69-73
123	EE	Dhruva R. Chakrabarti, Prithviraj Banerjee, Antonio Lain: Evaluation of Compiler and Runtime Library Approaches for Supporting Parallel Regular Applications. IPPS/SPDP 1998: 74-79

122	EE	Zhaoyun Xing, Prithviraj Banerjee: A parallel algorithm for zero skew clock tree routing. ISPD 1998: 118-123
121	EE	Venkatram Krishnaswamy, Prithviraj Banerjee: Parallel Compiled Event Driven VHDL Simulation. International Conference on Supercomputing 1998: 297-304
120	EE	Dhruva R. Chakrabarti, Nagaraj Shenoy, Alok N. Choudhary, Prithviraj Banerjee: An Efficient Uniform Run-time Scheme for Mixed Regular-irregular Applications. International Conference on Supercomputing 1998: 61-68
119	EE	Mahmut T. Kandemir, Alok N. Choudhary, Nagaraj Shenoy, Prithviraj Banerjee, J. Ramanujam: A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests. International Conference on Supercomputing 1998: 69-76
118	EE	Mahmut T. Kandemir, J. Ramanujam, Alok N. Choudhary, Prithviraj Banerjee: A Loop Transformation Algorithm Based on Explicit Data Layout Representation for Optimizing Locality. LCPC 1998: 34-50
117	EE	Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Prithviraj Banerjee: Improving Locality Using Loop and Data Transformations in an Integrated Framework. MICRO 1998: 285-297
116	EE	Gagan Hasteer, Anmol Mathur, Prithviraj Banerjee: Efficient equivalence checking of multi-phase designs using phase abstraction and retiming. ACM Trans. Design Autom. Electr. Syst. 3(4): 600-625 (1998)
115		Gagan Hasteer, Prithviraj Banerjee: A Parallel Algorithm for State Assignment of Finite State Machines. IEEE Trans. Computers 47(2): 242-246 (1998)
		1997
114	EE	Gagan Hasteer, Anmol Mathur, Prithviraj Banerjee: An Efficient Assertion Checker for Combinational Properties. DAC 1997: 734-739
113		John A. Chandy, Prithviraj Banerjee: A Parallel Circuit-Partitioned Algorithm for Timing Driven Cell Placement. ICCD 1997: 621-627
112	EE	Venkatram Krishnaswamy, Gagan Hasteer, Prithviraj Banerjee: Load Balancing and Workload Minimization Of Overlapping Parallel Tasks. ICPP 1997: 272-279
111	EE	Dilip Krishnaswamy, Prithviraj Banerjee: Exploiting task and data parallelism in parallel Hough and Radon transforms. ICPP 1997: 441-
110	EE	Zhaoyun Xing, John A. Chandy, Prithviraj Banerjee: Parallel Global Routing Algorithms for Standard Cells. IPPS 1997: 527-
109	EE	Sumit Roy, Prithviraj Banerjee: A Comparison of Parallel Approaches for Algebraic Factorization in Logic Synthesis. IPPS 1997: 665-671
108	EE	John G. Holm, John A. Chandy, Steven Parkes, Sumit Roy, Venkatram Krishnaswamy, Gagan Hasteer, Prithviraj Banerjee: Performance Evaluation of Message-Driven Parallel VLSI CAD Applications on General Purpose Multiprocessors. International Conference on Supercomputing 1997: 172-179
107	EE	Dilip Krishnaswamy, Prithviraj Banerjee, Elizabeth M. Rudnick, Janak H. Patel: Asynchronous Parallel Algorithms for Test Set Partitioned Fault Simulation. Workshop on Parallel and Distributed Simulation 1997: 30-37
106	EE	Shankar Ramaswamy, Sachin Sapatneker, Prithviraj Banerjee: A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. IEEE Trans. Parallel Distrib. Syst. 8(11): 1098-1116 (1997)

105		Gagan Hasteer, Prithviraj Banerjee: Simulated Annealing Based Parallel State Assignment of Finite State Machines. J. Parallel Distrib. Comput. 43(1): 21-35 (1997)
1996		
104		Vamsi Boppana, Prashant Saxena, Prithviraj Banerjee, W. Kent Fuchs, C. L. Liu: A Parallel Algorithm for the Technology Mapping of LUT-Based FPGAs. Euro-Par, Vol. I 1996: 828-831
103		Amber Roy-Chowdhury, Prithviraj Banerjee: Compiler-Assisted Generation of Error-Detecting Parallel Programs. FTCS 1996: 360-369
102	EE	Pradeep Prabhakaran, Prithviraj Banerjee: Parallel Algorithms for Force Directed Scheduling of Flattened and Hierarchical Signal Flow Graphs. ICCD 1996: 66-71
101		Gagan Hasteer, Prithviraj Banerjee: A Parallel Algorithm for State Assignment of Finite State Machines. ICPP, Vol. 2 1996: 37-45
100		Daniel J. Palermo, Ernesto Su, Eugene W. Hodges IV, Prithviraj Banerjee: Compiler Support for Privatization on Distributed-Memory Machines. ICPP, Vol. 3 1996: 17-24
99	EE	Shankar Ramaswamy, Eugene W. Hodges IV, Prithviraj Banerjee: Compiling MATLAB Programs to ScaLAPACK: Exploiting Task and Data Parallelism. IPPS 1996: 613-619
98		John A. Chandy, Steven Parkes, Prithviraj Banerjee: Distributed Object Oriented Data Structures and Algorithms for VLSI CAD. IRREGULAR 1996: 147-158
97	EE	Antonio Lain, Prithviraj Banerjee: Compiler Support for Hybrid Irregular Accesses on Multicomputers. International Conference on Supercomputing 1996: 1-9
96		Daniel J. Palermo, Eugene W. Hodges IV, Prithviraj Banerjee: Interprocedural Array Redistribution Data-Flow Analysis. LCPC 1996: 435-449
95	EE	Venkatram Krishnaswamy, Prithviraj Banerjee: Actor Based Parallel VHDL Simulation Using Time Warp. Workshop on Parallel and Distributed Simulation 1996: 135-142
94		Amber Roy-Chowdhury, Prithviraj Banerjee: Algorithm-Based Fault Location and Recovery for Matrix Computations on Multiprocessor Systems. IEEE Trans. Computers 45(11): 1239-1247 (1996)
93		Amber-Roy Chowdhury, Prithviraj Banerjee: A New Error Analysis Based Method for Tolerance Computation for Algorithm-Based Checks. IEEE Trans. Computers 45(2): 238-243 (1996)
92		Amber Roy-Chowdhury, Nikolas Bellas, Prithviraj Banerjee: Algorithm-Based Error Detection Schemes for Iterative Solution of Partial Differential Equations. IEEE Trans. Computers 45(4): 394-407 (1996)
91		V. S. S. Nair, Jacob A. Abraham, Prithviraj Banerjee: Efficient Techniques for the Analysis of Algorithm-Based Fault Tolerance (ABFT) Schemes. IEEE Trans. Computers 45(4): 499-503 (1996)
90		Ky MacPherson, Prithviraj Banerjee: Parallel Algorithms for VLSI Layout Verification. J. Parallel Distrib. Comput. 36(2): 156-172 (1996)
89		Daniel J. Palermo, Eugene W. Hodges IV, Prithviraj Banerjee: Dynamic Data Partitioning for Distributed-Memory Multicomputers. J. Parallel Distrib. Comput. 38(2): 158-175 (1996)
88		Shankar Ramaswamy, Barbara Simons, Prithviraj Banerjee: Optimizations for Efficient Array Redistribution on Distributed Memory Multicomputers. J. Parallel Distrib. Comput. 38(2): 217-228 (1996)
1995		

87		Michael Peercy, Prithviraj Banerjee: Software Schemes of Reconfiguration and Recovery in Distributed Memory Multicomputers Using the Actor Model. FTCS 1995: 479-488
86	EE	Steven Parkes, Prithviraj Banerjee, Janak Patel: A parallel algorithm for fault simulation based on PROOFS . ICCD 1995: 616-
85	EE	Kaushik De, John A. Chandy, Sumit Roy, Steven Parkes, Prithviraj Banerjee: Parallel algorithms for logic synthesis using the MIS approach. IPPS 1995: 579-585
84	EE	Antonio Lain, Prithviraj Banerjee: Exploiting spatial regularity in irregular iterative applications. IPPS 1995: 820-826
83	EE	Ernesto Su, Antonio Lain, Shankar Ramaswamy, Daniel J. Palermo, Eugene W. Hodges IV, Prithviraj Banerjee: Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-memory Multicomputers. International Conference on Supercomputing 1995: 424-433
82		Daniel J. Palermo, Prithviraj Banerjee: Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. LCPC 1995: 392-406
81		Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, Ernesto Su: The Paradigm Compiler for Distributed-Memory Multicomputers. IEEE Computer 28(10): 37-47 (1995)
80		Shankar Ramaswamy, Prithviraj Banerjee: Simultaneous Allocation and Scheduling Using Convex Programming Techniques. Parallel Processing Letters 5: 587-598 (1995)
		1994
79	EE	Steven Parkes, Prithviraj Banerjee, Janak H. Patel: ProperHITEC: A Portable, Parallel, Object-Oriented Approach to Sequential Test Generation. DAC 1994: 717-721
78		Amber Roy-Chowdhury, Prithviraj Banerjee: Algorithm-Based Fault Location and Recovery for Matrix Computations. FTCS 1994: 38-47
77		Daniel J. Palermo, Ernesto Su, John A. Chandy, Prithviraj Banerjee: Communication Optimizations Used in the PARADIGM Compiler for Distributed Memory Multicomputers. ICPP 1994: 1-10
76		Shankar Ramaswamy, Sachin S. Sapatnekar, Prithviraj Banerjee: A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers. ICPP 1994: 116-125
75		Kaushik De, Prithviraj Banerjee: Parallel Logic Synthesis Using Partitioning. ICPP (3) 1994: 135-142
74		Ernesto Su, Daniel J. Palermo, Prithviraj Banerjee: Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers. IFIP PACT 1994: 123-132
73		Sungho Kim, Prithviraj Banerjee, Balkrishna Ramkumar, Steven Parkes, John A. Chandy: ProperPLACE: A Portable Parallel Algorithm for Standard Cell Placement. IPPS 1994: 932-941
72	EE	Antonio Lain, Prithviraj Banerjee: Techniques to overlap computation and communication in irregular iterative applications. International Conference on Supercomputing 1994: 236-245
71	EE	Steven Parkes, John A. Chandy, Prithviraj Banerjee: A library-based approach to portable, parallel, object-oriented programming: interface, implementation, and application. SC 1994: 69-78
70		Prithviraj Banerjee, Michael Peercy: Design and Evaluation of Hardware Strategies for Reconfiguring Hypercubes and Meshes Under Faults. IEEE Trans. Computers 43(7): 841-848

		(1994)
1993		
69	EE	Vivek Chickermane, Elizabeth M. Rudnick, Prithviraj Banerjee, Janak H. Patel: Non-Scan Design-for-Testability Techniques for Sequential Circuits. DAC 1993: 236-241
68		Amber Roy-Chowdhury, Prithviraj Banerjee: Tolerance Determination for Algorithm-Based Checks Using Simplified Error Analysis Techniques. FTCS 1993: 290-298
67		Amber Roy-Chowdhury, Prithviraj Banerjee: A Fault-Tolerant Parallel Algorithm for Iterative Solution of the Laplace Equation. ICPP 1993: 133-140
66		Shankar Ramaswamy, Prithviraj Banerjee: Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler. ICPP 1993: 134-138
65		John A. Chandy, Prithviraj Banerjee: Reliability Evalutaion of Disk Array Architectures. ICPP 1993: 263-267
64		Ernesto Su, Daniel J. Palermo, Prithviraj Banerjee: Automating Parallelization of Regular Computations for Distributed-Memory. ICPP 1993: 30-38
63		Balkrishna Ramkumar, Prithviraj Banerjee: A Portable Parallel Algorithm for VLSI Circuit Extraction. IPPS 1993: 434-438
62	EE	Manish Gupta, Prithviraj Banerjee: PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. International Conference on Supercomputing 1993: 87-96
61		Chieng-Fai Lim, Prithviraj Banerjee, Kaushik De, Saburo Muroga: A Shared Memory Parallel Algorithm for Logic Synthesis. VLSI Design 1993: 317-322
60		A. L. Narasimha Reddy, John Chandy, Prithviraj Banerjee: Design and Evaluation of Gracefully Degradable Disk Arrays. J. Parallel Distrib. Comput. 17(1-2): 28-40 (1993)
1992		
59	EE	Sungho Kim, Prithviraj Banerjee, Vivek Chickermane, Janak H. Patel: APT: An Area-Performance-Testability Driven Placement Algorithm. DAC 1992: 141-146
58		Michael Peercy, Prithviraj Banerjee: Design and Analysis of Software Reconfiguration Strategies for Hypercube Multicomputers under Multiple Faults. FTCS 1992: 448-455
57	EE	Balkrishna Ramkumar, Prithviraj Banerjee: Portable parallel test generation for sequential circuits. ICCAD 1992: 220-223
56	EE	Kaushik De, Balkrishna Ramkumar, Prithviraj Banerjee: ProperSYN: a portable parallel algorithm for logic synthesis. ICCAD 1992: 412-416
55		Balkrishna Ramkumar, Prithviraj Banerjee: ProperCAd: A Portable Object-Oriented Parallel Environment for VLSI CAD. ICCD 1992: 544-548
54		John G. Holm, Prithviraj Banerjee: Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions. ICPP (1) 1992: 192-195
53	EE	Manish Gupta, Prithviraj Banerjee: A methodology for high-level synthesis of communication on multicomputers. ICS 1992: 357-367
52		Manish Gupta, Prithviraj Banerjee: Compile-Time Estimation of Communication Costs on Multicomputers. IPPS 1992: 470-475
51		Krishna P. Belkhale, Prithviraj Banerjee: Reconfiguration Strategies for VLSI Processor Arrays and Trees Using a Modified Diogenes Approach. IEEE Trans. Computers 41(1): 83-96 (1992)

50		Krishna P. Belkhale, Prithviraj Banerjee: Parallel Algorithms for Geometric Connected Component Labeling on a Hypercube Multiprocessor. IEEE Trans. Computers 41(6): 699-709 (1992)
49	EE	Manish Gupta, Prithviraj Banerjee: Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. IEEE Trans. Parallel Distrib. Syst. 3(2): 179-193 (1992)
48	EE	Jiun-Ming Hsu, Prithviraj Banerjee: Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer. IEEE Trans. Parallel Distrib. Syst. 3(4): 451-464 (1992)
1991		
47	EE	Srinivas Patil, Prithviraj Banerjee, Janak H. Patel: Parallel Test Generation for Sequential Circuits on General-Purpose Multiprocessors. DAC 1991: 155-159
46		A. L. Narasimha Reddy, Prithviraj Banerjee: Gracefully Degradable Disk Arrays. FTCS 1991: 401-409
45		Vijay Balasubramanian, Prithviraj Banerjee: CRAFT: Compiler-Assisted Algorithm-Based Fault Tolerance in Distributed Memory Multiprocessors. ICPP (1) 1991: 501-504
44		Jiun-Ming Hsu, Prithviraj Banerjee: Performance Evaluation of Hardware Support for Message Passing in Distributed Memory Multicomputers. ICPP (1) 1991: 604-607
43		A. L. Narasimha Reddy, Prithviraj Banerjee, D. K. Chen: Compiler Support for Parallel I/O Operations. ICPP (2) 1991: 290-291
42		Krishna P. Belkhale, Prithviraj Banerjee: A Scheduling Algorithm for Parallelizable Dependent Tasks. IPPS 1991: 500-506
41		Sungho Kim, Prithviraj Banerjee, Srinivas Patil: A Layout Driven Design for Testability Technique for MOS VLSI Circuits. ITC 1991: 157-165
40		Kaushik De, Prithviraj Banerjee: Logic Partitioning and Resynthesis for Testability. ITC 1991: 906-915
1990		
39	EE	Ralph-Michael Kling, Prithviraj Banerjee: Optimization by Simulated Evolution with Applications to Standard Cell Placement. DAC 1990: 20-25
38	EE	Randall J. Brouwer, Prithviraj Banerjee: PHIGURE: A Parallel Hierarchical Global Router. DAC 1990: 650-653
37		Krishna P. Belkhale, Prithviraj Banerjee: A Parallel Algorithm for Hierarchical Circuit Extraction. ICCAD 1990: 236-239
36		David Blaauw, Robert Mueller-Thuns, Daniel G. Saab, Prithviraj Banerjee, Jacob A. Abraham: SNEL: A Switch-Level Simulator Using Multiple Levels of Functional Abstraction. ICCAD 1990: 66-69
35		Jiun-Ming Hsu, Prithviraj Banerjee: Hardware Support for Message Routing in a Distributed Memory Multicomputer. ICPP (1) 1990: 508-515
34		Krishna P. Belkhale, Prithviraj Banerjee: An Approximate Algorithm for the Partitionable Independent Task Scheduling Problem. ICPP (1) 1990: 72-75
33		Krishna P. Belkhale, Prithviraj Banerjee: Geometric Connected Component Labeling on Distributed Memory Multicomputers. ICPP (3) 1990: 291-294
32		Jiun-Ming Hsu, Prithviraj Banerjee: Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer. ISCA 1990: 260-

		269
31		A. L. Narasimha Reddy, Prithviraj Banerjee: A Study of I/O Behavior of Perfect Benchmarks on a Multiprocessor. ISCA 1990: 312-321
30	EE	Jiun-Ming Hsu, Prithviraj Banerjee: A message passing coprocessor for distributed memory multicomputers. SC 1990: 720-729
29		A. L. Narasimha Reddy, Prithviraj Banerjee: Algorithms-Based Fault Detection for Signal Processing Applications. IEEE Trans. Computers 39(10): 1304-1308 (1990)
28		Vijay Balasubramanian, Prithviraj Banerjee: Compiler-Assisted Synthesis of Algorithm-Based Checking in Multiprocessors. IEEE Trans. Computers 39(4): 436-446 (1990)
27		Prithviraj Banerjee, Joseph T. Rahmeh, Craig B. Stunkel, V. S. S. Nair, Kaushik Roy, Vijay Balasubramanian, Jacob A. Abraham: Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor. IEEE Trans. Computers 39(9): 1132-1145 (1990)
26	EE	Prithviraj Banerjee, Mark Howard Jones, Jeff S. Sargent: Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1): 91-106 (1990)
25	EE	A. L. Narasimha Reddy, Prithviraj Banerjee: Design, Analysis, and Simulation of I/O Architectures for Hypercube. IEEE Trans. Parallel Distrib. Syst. 1(2): 140-151 (1990)
24	EE	Vijay Balasubramanian, Prithviraj Banerjee: Tradeoffs in the Design of Efficient Algorithm-Based Error Detection Schemes for Hypercube Multiprocessors. IEEE Trans. Software Eng. 16(2): 183-196 (1990)
		1989
23	EE	Srinivas Patil, Prithviraj Banerjee: A Parallel Branch and Bound Algorithm for Test Generation. DAC 1989: 339-343
22	EE	Jeff S. Sargent, Prithviraj Banerjee: A Parallel Row-based Algorithm for Standard Cell Placement with Integrated Error Control. DAC 1989: 590-593
21		A. L. Narasimha Reddy, Prithviraj Banerjee: Performance Evaluation of Multiple-Disk I/O Systems. ICPP (1) 1989: 315-318
20		Robert B. Mueller-Thuns, David McFarland, Prithviraj Banerjee: Algorithm-Based Fault Tolerance for Adaptive Least Squares Lattice Filtering on a Hypercube Multiprocessor. ICPP (3) 1989: 177-180
19	EE	A. L. Narasimha Reddy, Prithviraj Banerjee: I/O issues for hypercubes. ICS 1989: 72-81
18		Vijay Balasubramanian, Prithviraj Banerjee: Algorithm-based Error Detection for Signal Processing Applications on a Hypercube Multiprocessor. IEEE Real-Time Systems Symposium 1989: 134-143
17		Srinivas Patil, Prithviraj Banerjee: Fault Partitioning Issues in an Integrated Parallel Test Generation/Fault Simulation Environment. ITC 1989: 718-726
16		Prithviraj Banerjee, Abhijeet Dugar: The Design, Analysis and Simulation of a Fault-Tolerant Interconnection Network Supporting the Fetch-and-Add Primitive. IEEE Trans. Computers 38(1): 30-46 (1989)
15		A. L. Narasimha Reddy, Prithviraj Banerjee: An Evaluation of Multiple-Disk I/O Systems. IEEE Trans. Computers 38(12): 1680-1690 (1989)
		1988
14		A. L. Narasimha Reddy, Prithviraj Banerjee: I/O Embedding in Hypercubes. ICPP (1) 1988: 331-338

13		Prithviraj Banerjee: The Cubical Ring Connected Cycles: A Fault-Tolerant Parallel Computation Network. IEEE Trans. Computers 37(5): 632-636 (1988)
12		Douglas B. West, Prithviraj Banerjee: On the Construction of Communication Networks Satisfying Bounded Fan-In of Service Ports. IEEE Trans. Computers 37(9): 1148-1151 (1988)
1987		
11	EE	Ralph-Michael Kling, Prithviraj Banerjee: ESP: A New Standard Cell Placement Package Using Simulated Evolution. DAC 1987: 60-66
10	EE	M. Jones, Prithviraj Banerjee: Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube. DAC 1987: 807-813
9	EE	A. L. Narasimha Reddy, Prithviraj Banerjee: A Fault Secure Dictionary Machine. ICDE 1987: 104-110
8		Vijay Balasubramanian, Prithviraj Banerjee: A Fixed Size Array Processor for Computing the Fast Fourier Transform. IEEE Real-Time Systems Symposium 1987: 36-43
7		Vijay Balasubramanian, Prithviraj Banerjee: A Fault Tolerant Massively Parallel Processing Architecture. J. Parallel Distrib. Comput. 4(4): 363-383 (1987)
1986		
6		Prithviraj Banerjee, Abhijeet Dugar: A Fault-Tolerant Interconnection Network Supporting the Fetch-And-Add Primitive. ICPP 1986: 327-334
5		Vijay Balasubramanian, Prithviraj Banerjee: RECBAR : A Reconfigurable Massively Parallel Processing Architecture. ICPP 1986: 390-393
4		Prithviraj Banerjee, Jacob A. Abraham: A Probabilistic Model of Algorithm-Based Fault Tolerance in Array Processors for Real-Time Systems. IEEE Real-Time Systems Symposium 1986: 72-78
3		Prithviraj Banerjee, Jacob A. Abraham: Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems. IEEE Trans. Computers 35(4): 296-306 (1986)
1984		
2		Prithviraj Banerjee, Jacob A. Abraham: Fault-Secure Algorithms for Multiple-Processor Systems. ISCA 1984: 279-287
1983		
1		Prithviraj Banerjee, Jacob A. Abraham: Generating Tests for Physical Failures in MOS Logic Circuits. ITC 1983: 554-559

DBLP: [Home | Search: Author, Title | Conferences | Journals]
Michael Ley (ley@uni-trier.de) Fri Jan 30 16:32:41 2004

Copy/Holding information

Collection	Call No.	Copy	Status
EIC for TC2600 Reserve	QA297 .H296 1998	c.3	Available
EIC for TC 2100	QA297 .H296 1998	c.1	Available
EIC for TC 2600	QA297 .H296 1998	c.2	Checked out

NPL Virtual Library

My List - 0 Help

Search

Main Search | Advanced Keyword Search | Search History

Search: Subject Keyword matlab Refine Search

You're searching: **Scientific and Technical Information Center**

Sort by: Publication date

Limit by: EIC for TC 2100

Search Results

17 titles matched: **matlab**

Next

1. Introduction to MATLAB for engineers and scientists / Delores M. Etter.
   by *Etter, D. M.*
   Prentice Hall, c1996.
   Call No.: TA345 .E873 1996
   Add to my list

2. The student edition of MATLAB : version 5, user's guide / The MathWorks, Inc. ; by Duane Hanselman and Bruce Littlefield.
   by *Hanselman, Duane C.*
   Prentice Hall, c1997.
   Call No.: QA297 .S8436 1997
   Add to my list

3. Signal processing toolbox for use with MATLAB / by Thomas P. Krauss, Loren Shure, and John N. Little.
   by *Krauss, Thomas P.*
   The MathWorks, 1998.
   Call No.: TK5102.9 .K72x 1998
   Add to my list

4. SIMULINK dynamic system simulation for MATLAB : modeling, simulation, implementation Writing S-functions.
   by *MathWorks, Inc.*
   MathWorks, Inc., c1998.
   Call No.: TA345.5.M12 S56x 1998
   Add to my list

5. Power System Blockset for use with Simulink® : user's guide / Hydro-Québec ; TEQSIM International.

by *TEQSIM International.*
MathWorks Inc., c1998.
Call No.: QA76.76.C672 P6 1998

Add to my list

6. Fixed-Point Blockset for use with Simulink® : user's guide.
by *MathWorks, Inc.*
MathWorks, c1998.
Call No.: QA76.76.C672 F5 1998

Add to my list

7. Mastering MATLAB 5 : a comprehensive tutorial and reference / Duane Hanselman, Bruce Littlefield.
by *Hanselman, Duane C.*
Prentice Hall, c1998.
Call No.: QA297 .H296 1998

Add to my list

8. Digital signal processing : a computer-based approach / Sanjit K. Mitra.
by *Mitra, Sanjit Kumar.*
McGraw-Hill, c1998.
Call No.: TK5102.9 .M57 1998

Add to my list

9. Computer-based exercises for signal processing using MATLAB 5 / James H. McClellan ... [et al.]
by *McClellan, James H., 1947-*
Prentice Hall, c1998.
Call No.: TK5102.9 .C567 1998

Add to my list

10. DSP blockset for use with SIMULINK : user's guide.
by *MathWorks, Inc.*
MathWorks, Inc., c1999.
Call No.: TK5102.5 .D761x 1999

Add to my list

1 2 Next

Add/Remove MyList (max=100, ie. 1,2 5-20) 1-10 Add Remove

iPac 2.03.01

US 20040019883A1

(19) United States
(12) Patent Application Publication
Banerjee et al.

(10) Pub. No.: US 2004/0019883 A1
(43) Pub. Date: Jan. 29, 2004

(54) METHOD AND APPARATUS FOR AUTOMATICALLY GENERATING HARDWARE FROM ALGORITHMS DESCRIBED IN MATLAB

(75) Inventors: **Prithviraj Banerjee**, Glenview, IL (US); **Alok Choudhary**, Chicago, IL (US); **Malay Haldar**, Evanston, IL (US); **Anshuman Nayak**, Evanston, IL (US)

Correspondence Address:
THE LAW OFFICE OF DEEPTI PANCHAWAGH-JAIN
3039 CALLE DE LAS ESTRELLA
SAN JOSE, CA 95148 (US)

(73) Assignee: NORTHWESTERN UNIVERSITY

(21) Appl. No.: 09/770,541

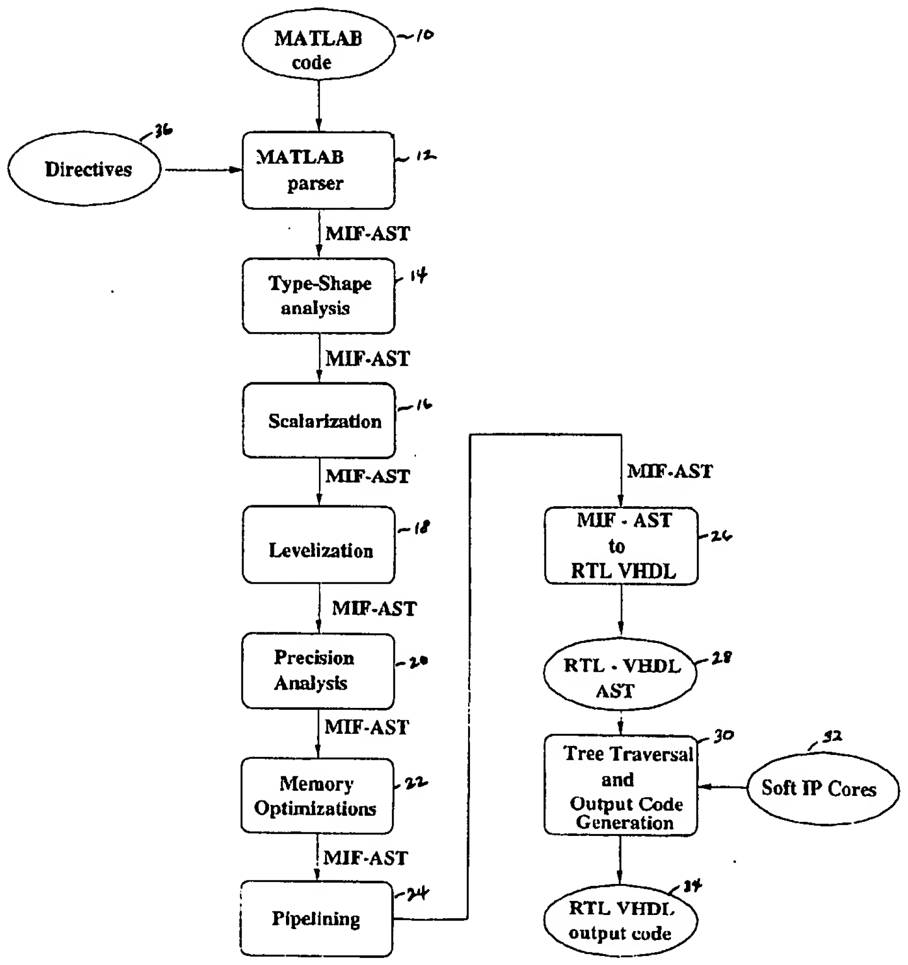
(22) Filed: Jan. 26, 2001

Publication Classification

(51) Int. Cl.^{7} G06F 9/45
(52) U.S. Cl. 717/152; 717/150; 717/159

(57) ABSTRACT

Digital circuit is synthesized from algorithm described in the MATLAB programming language. A MATLAB program is compiled into RTL-VHDL, which is synthesizable using system-specific tools to develop ASIC or FPGA configuration. Intermediate transformations and optimizations are performed to obtain highly optimized description in RTL-VHDL or RTL Verilog of given MATLAB program. Optimizations include levelization, scalarization, pipelining, type-shape analysis, memory optimizations, precision analysis and scheduling.



Synthesis flow.

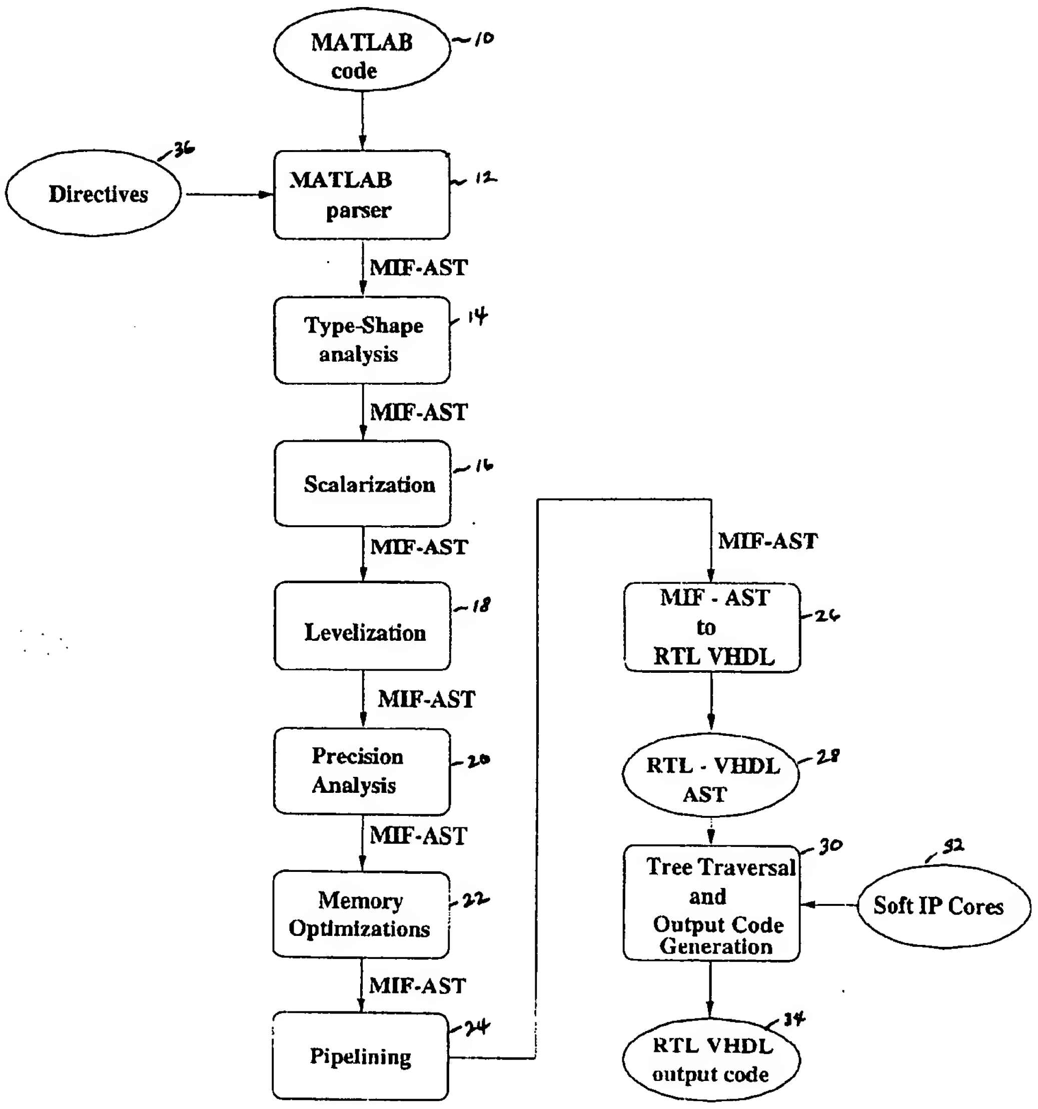
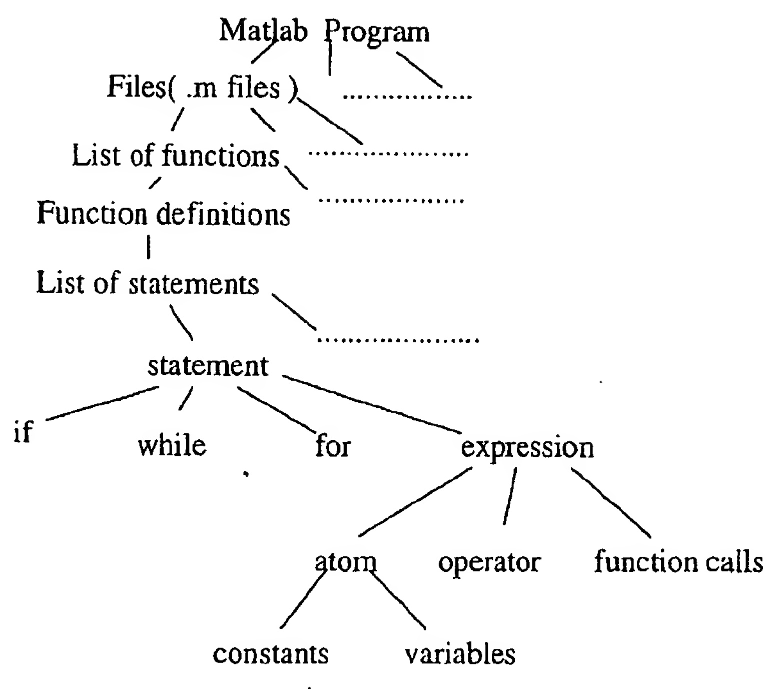


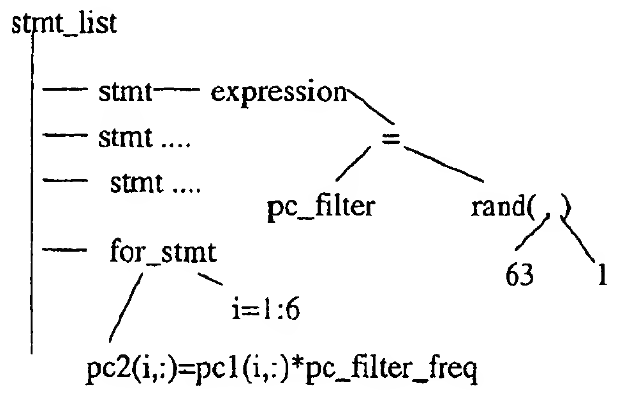
Figure 1: Synthesis flow.



(a)

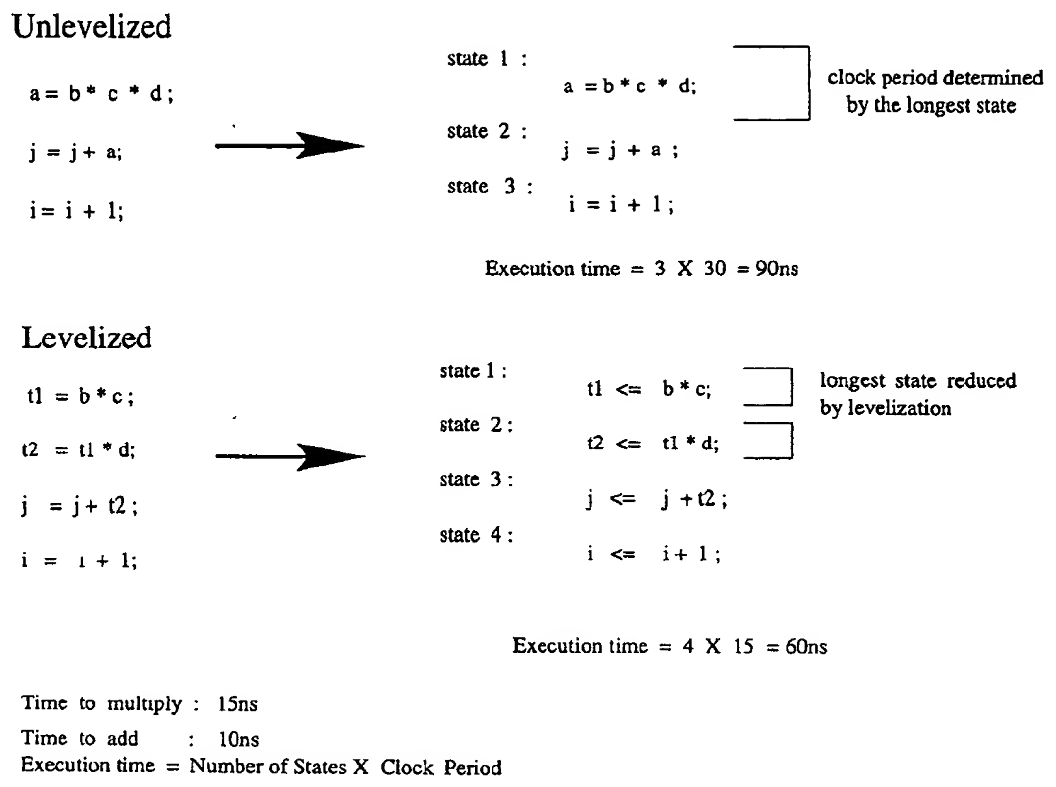
```
pc_filter = rand(63,1);
pc_filter_time = zeros9512,1);
pc_filter_frq = fft(pc_filter_time);
for i = 1:6
 pc2(i,:) = pc1(i,:)*pc_filter_freq
end;
```

(b)



(c)

Figure 2: Abstract Syntax Tree: (a) The hierarchy captured by the formal grammar (b) A sample code snippet (c) Abridged syntax tree for the code snippet.


Unlevelized
clock period determined by the longest state
Levelized
longest state reduced by levelization

```
a= b* c * d;
j = j+ a;
i= i + 1;
```

```
state 1 :
        a =b * c * d;
state 2 :
        j = j + a ;
state 3 :
        i = i + 1 ;
```

Execution time = 3 X 30 = 90ns

```
t1 = b * c;
t2 = t1 * d;
j = j+ t2;
i = i + 1;
```

```
state 1 :
        t1 <= b * c;
state 2 :
        t2 <= t1 * d;
state 3 :
        j <= j +t2 ;
state 4 :
        i <= i+ 1 ;
```

Execution time = 4 X 15 = 60ns

Time to multiply : 15ns
Time to add : 10ns
Execution time = Number of States X Clock Period

Figure 3: Levelization improves clock period.

```
a = 1 ;
b = 1 ;
c = a + b ;
c = c + 1 ;
```

(a)

```
when state 1 =>  a  <= 1 ;
                next_state <= state 2 ;
when state 2 =>  b  <= 1 ;
                next_state <= state 3 ;
when state 3 =>  c  <= a + b ;
                next_state <= state 4 ;
when state 4 =>  c  <= c + 1 ;
```

(b)

Figure 4: (a) Simple MATLAB statements. (b) State machine that steps through the statements sequentially.

```
if( x )
  a = b + 1;
else
  a = b - 2 ;
end ;
c = a + 1 ;
```

(a)

```
when state 1 =>  if( x ) then
                   next_state <= state 2 ;
                 else
                   next_state <= state 3 ;
when state 2 =>  a <= b + 1 ;
                   next_state <= state 4 ;
when state 3 =>  a <= b - 2 ;
                   next_state <= state 4;
when state 4 =>
                 c <= a + 1 ;
```

(b)

Figure 5: (a) Conditional MATLAB code (b) State machine for Conditional Code.

```
for i = 1 : 256
    a = a + i ;
end ;
```

(a)

```
when state 1 => i <= 1 ;
                next_state <= state 2 ;
when state 2 => if( i <= 256 ) then
                  next_state <= state 3 ;
                else
                  next_state <= state 5;
                endif ;
when state 3 => a <= a + i ;
              next_state <= state 4 ;
when state 4 => i <= i + 1 ;
              next_state <= state 2 ;
when state 5 => [ next statement after loop ]
```

(b)

Figure 6: (a) A MATLAB *for* loop (b) State machine for *for* loop.

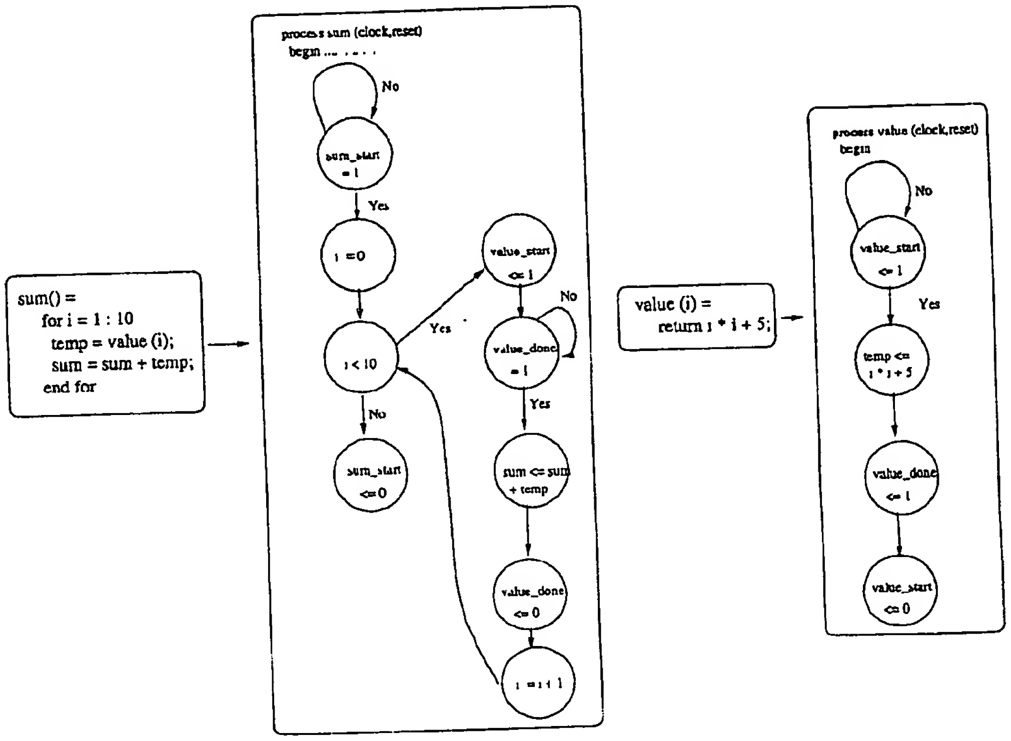


Figure 7: State machine representation of a function call in MATLAB

```
a[ i + 1 ] = b +  c ;
       |
  Levelization
       ↓

t1 =  b  +  c  ;
t2 =  i + 1;
a[ t2 ] = t1 ;
```

(a)

```
when state 1 => t1  <=  b + c ;
                   next_state <= state 2 ;
when state 2 => t2  <=  i  +  1 ;
                   next_state <= state 3 ;
when state 3 => mem_request  <=  '0' ;
                   mem_write_enable <= '0' ;
                   next_state <= state 4 ;
when state 4 => mem_address <= Base_a  + t2 ;
                   mem_data_out <= t1 ;
                   next_state <= state 5 ;
when state 5 =>  if( mem_grant = '0' ) then
                       next_state <= state 6;
                    else
                       next_state <= state 4 ;
                    endif;
```

(b)

Figure 8: (a) Array statement in MATLAB and its levelization (b) VHDL corresponding to the MATLAB code. The signals *mem_request*, *mem_data_out*, *mem_grant*, *mem_write_enable* and their particular states and assignments are specified in an external file read by the compiler. *Base_a* is a constant denoting the starting address of the array *a* in memory.

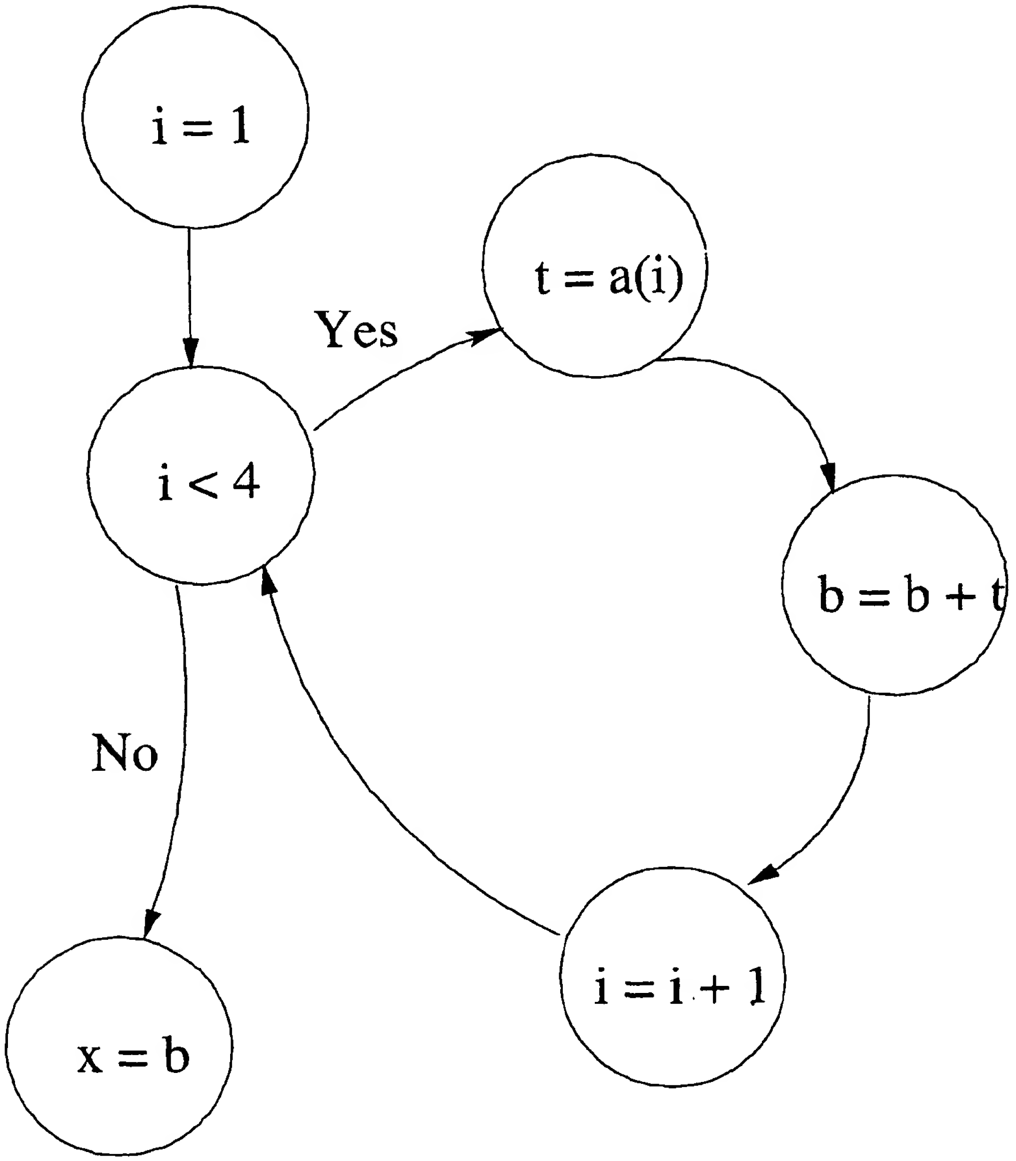


Figure 9: FSM Representation of a code section

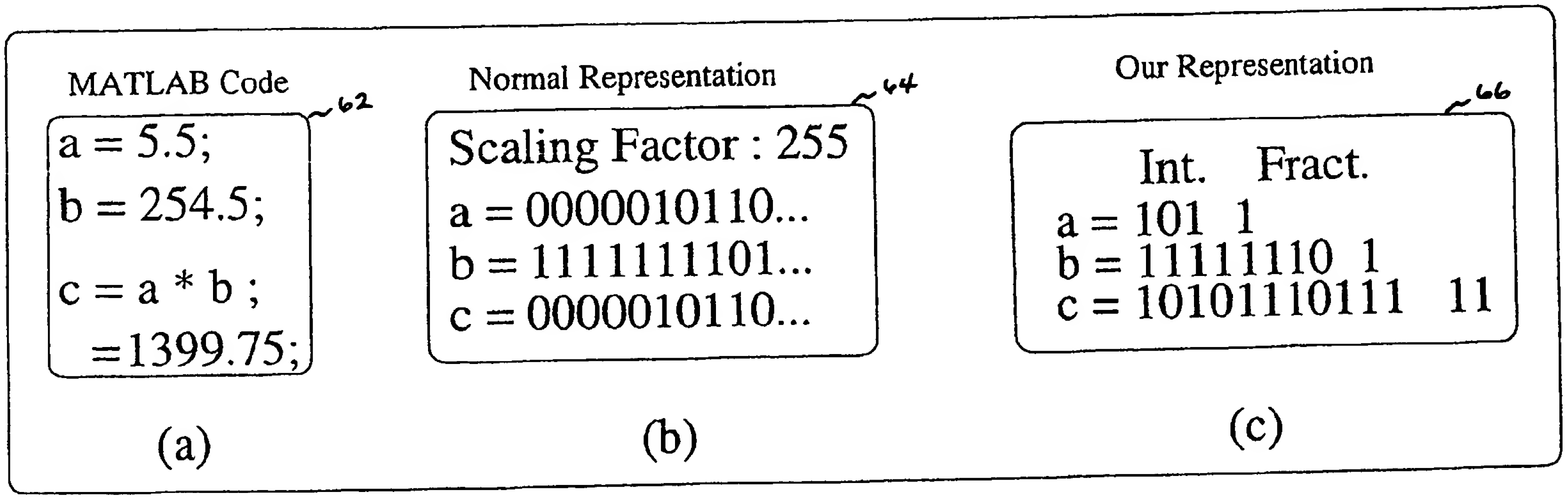
Algorithm 1 architecture ...

```
process ...

    if reset = '1' then
        ...
    elsif rising_edge(clock)
        case control is
            when state1 ⇒
                i := 1;
                control ⇐ state2;
            when state2 ⇒
                if (i < 4) then
                ....
                else
                ....
                endif;
            when state3 ⇒
                t := a(i);
                control ⇐ state4;
            when state4 ⇒
                b := b + t;
                control ⇐ state5;
            when state5 ⇒
                i := i + 1;
                control ⇐ state2;
```

Figure 10: VHDL code generated for the above FSM

over manual design or design at a low level of algorithm description (e.g., directly in VHDL).



```
a = 5.5;
b = 254.5;

c = a * b ;
  =1399.75;
```

```
Scaling Factor : 255
a = 0000010110...
b = 1111111101...
c = 0000010110...
```

```
     Int.    Fract.
a = 101       1
b = 11111110  1
c = 10101110111   11
```

Figure 11: Representation of Real Variables

```
for i = 1:1:20
    a[i+2] = b[i] + c[i+1];
end
```

$\longrightarrow$

```
for i = 1:4:20
    a[i+2] = b[i] + c[i+1];
    a[i+3] = b{i+1] + c[i+2];
    a[i+4] = b[i+2] + c[i+3];
    a[i+5] = b[i+3] + c[i+4];
end
```

Figure 12: Example showing loop unrolled for Memory Packing

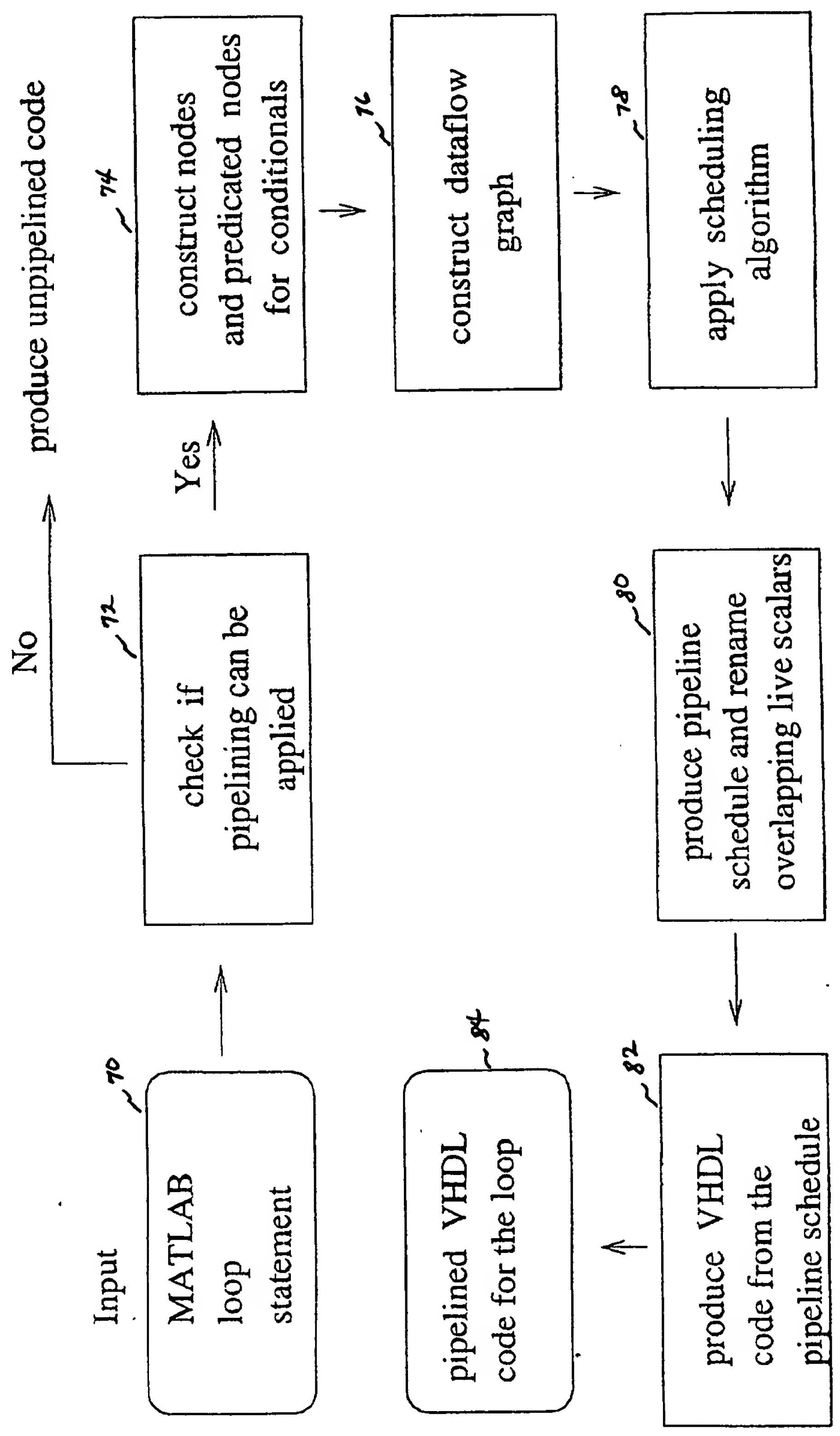
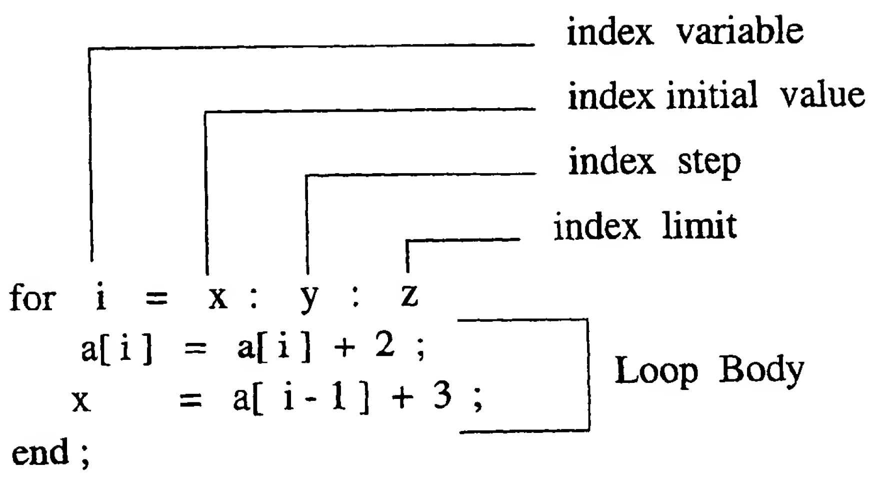
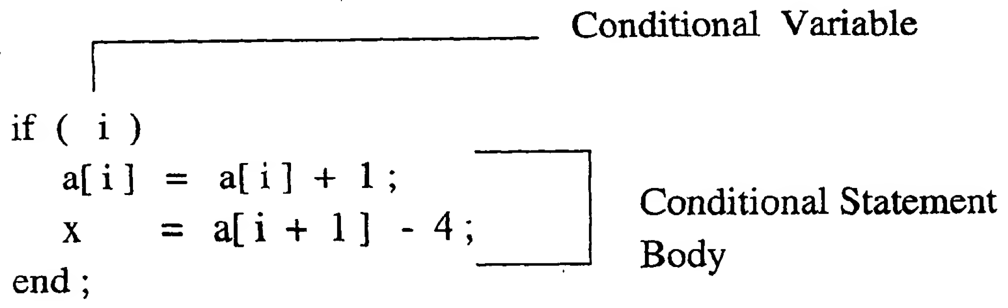

Input
MATLAB loop statement
70
check if pipelining can be applied
72
No
produce unpipelined code
Yes
construct nodes and predicated nodes for conditionals
74
construct dataflow graph
76
apply scheduling algorithm
78
produce pipeline schedule and rename overlapping live scalars
80
produce VHDL code from the pipeline schedule
82
pipelined VHDL code for the loop
84

Figure 13: Overall framework for pipelining optimizations

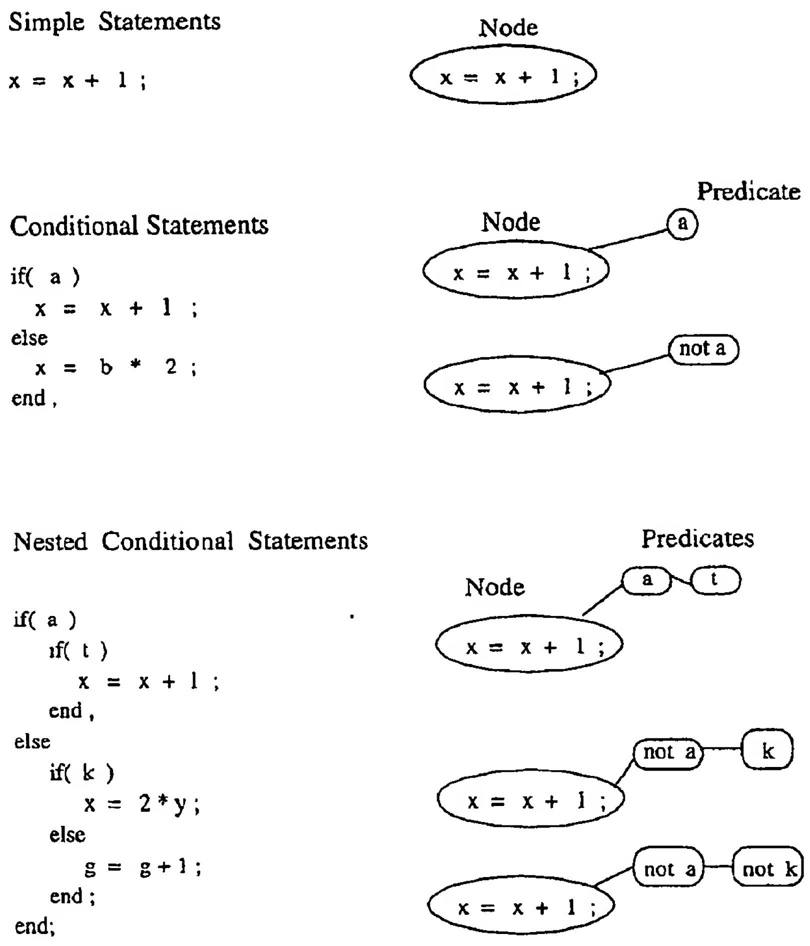


Loop Statement



Conditional Statement

Figure 14: Illustration of Terms used in pipelining framework


Simple Statements

```
x = x + 1 ;
```

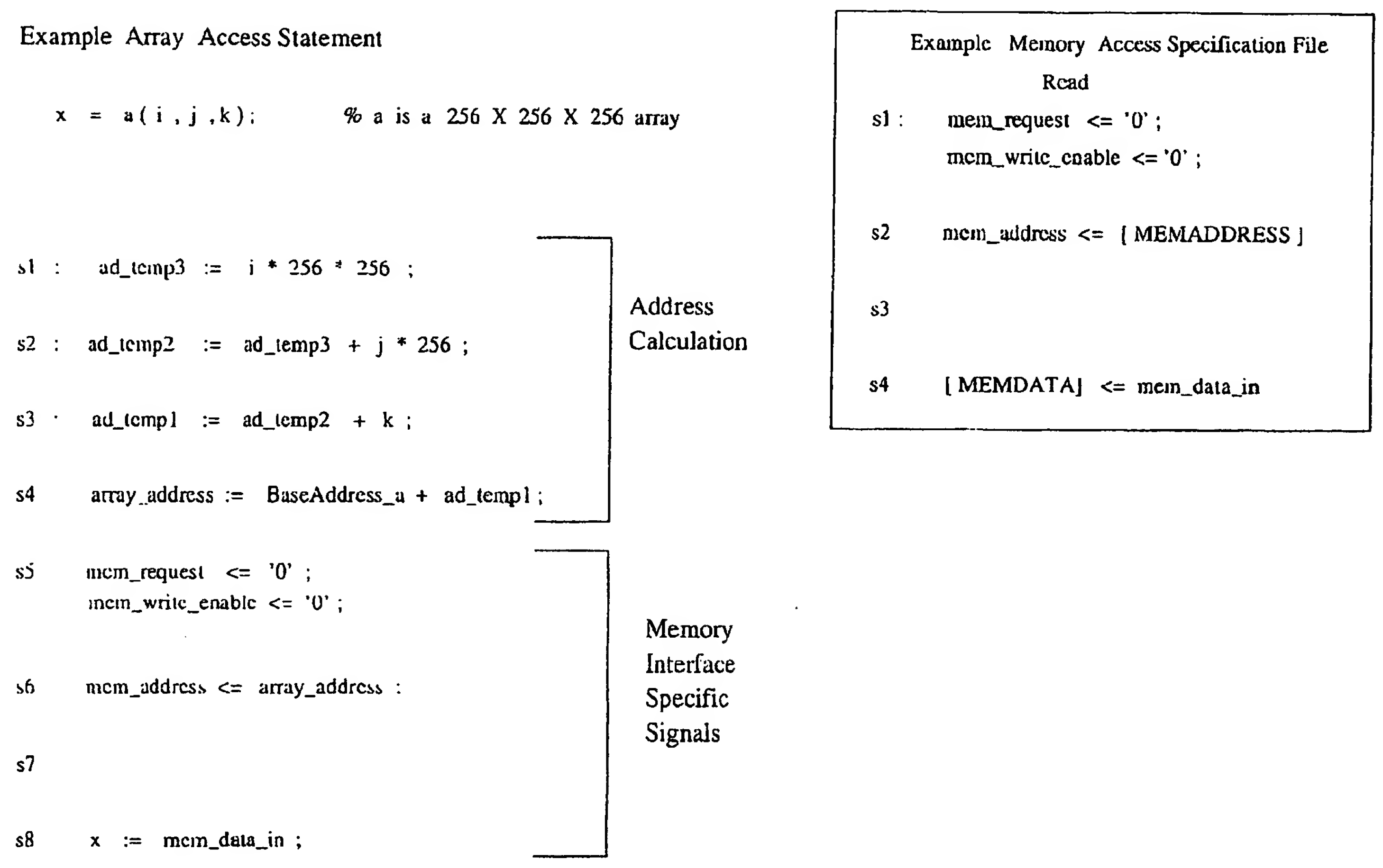
Conditional Statements

```
if( a )
  x = x + 1 ;
else
  x = b * 2 ;
end ,
```

Nested Conditional Statements

```
if( a )
   if( t )
      x = x + 1 ;
   end ,
else
   if( k )
      x = 2*y ;
   else
      g = g+1 ;
   end ;
end;
```

Figure 15: Illustration of Construction of nodes from MATLAB statements



Example Array Access Statement

```
x  =  a( i , j ,k );          % a is a 256 X 256 X 256 array

s1 :   ad_temp3  :=  i * 256 * 256 ;

s2 :  ad_temp2   :=  ad_temp3 + j * 256 ;

s3 ·  ad_temp1   :=  ad_temp2  + k ;

s4    array_address :=  BaseAddress_a + ad_temp1 ;

s5    mem_request  <=  '0' ;
      mem_write_enable <= '0' ;

s6    mem_address <=  array_address :

s7

s8    x  :=  mem_data_in ;
```

Example Memory Access Specification File

Read

```
s1 :    mem_request  <= '0' ;
        mem_write_enable <= '0' ;

s2      mem_address <=  [ MEMADDRESS ]

s3

s4      [ MEMDATA ]  <= mem_data_in
```

Figure 16: Example of node construction for array access statements

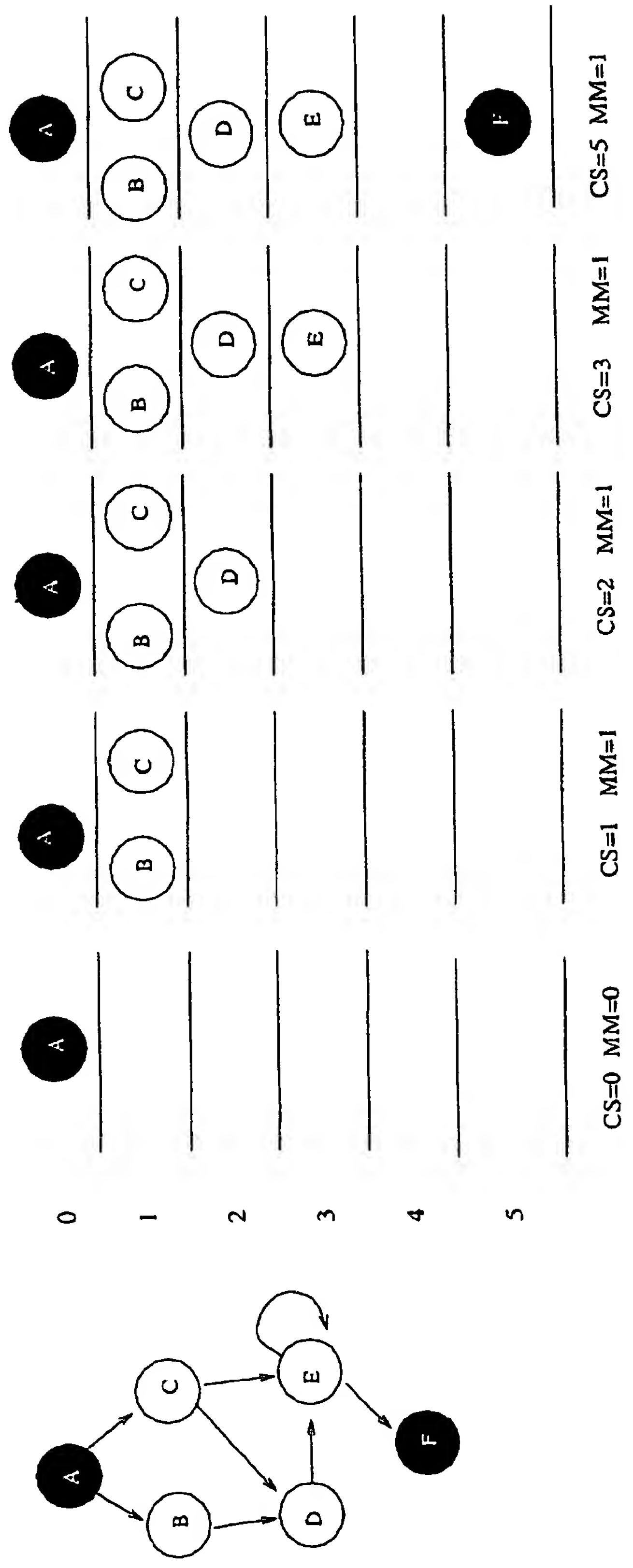
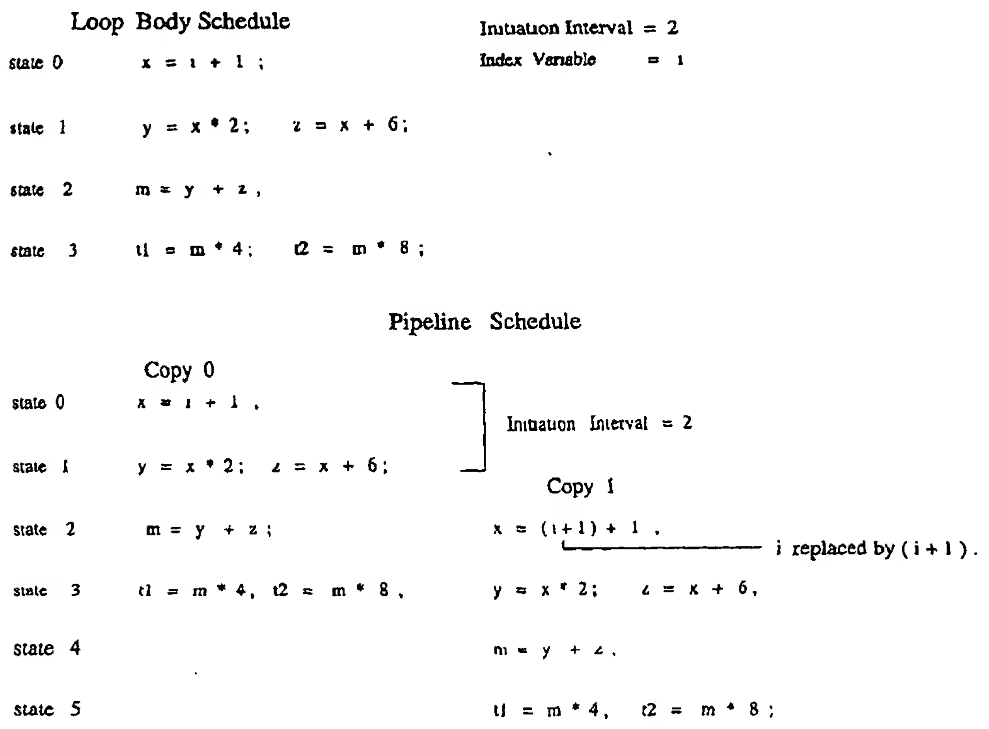


Figure 17: An illustration of pipeline method



```
        Loop Body Schedule                          Initiation Interval = 2
state 0     x = i + 1 ;                             Index Variable      = i

state 1     y = x * 2;   z = x + 6;

state 2     m = y + z ,

state 3     t1 = m * 4;   t2 = m * 8 ;

                          Pipeline  Schedule

            Copy 0
state 0     x = i + 1 ,                        ┐
                                               │ Initiation Interval = 2
state 1     y = x * 2;  z = x + 6;             ┘
                                                 Copy 1
state 2     m = y + z ;                        x = (i+1) + 1 .
                                                   └──────────── i replaced by ( i + 1 ) .
state 3     t1 = m * 4,  t2 = m * 8 ,          y = x * 2;   z = x + 6,

state 4                                        m = y + z .

state 5                                        t1 = m * 4,   t2 = m * 8 ;
```

Figure 18: Construction of pipeline schedule from loop body schedule

Loop Body Schedule

```
s1    x = i + 1 ;
s2    y = i * 2 ,
s3    k = x + 3 ,
```

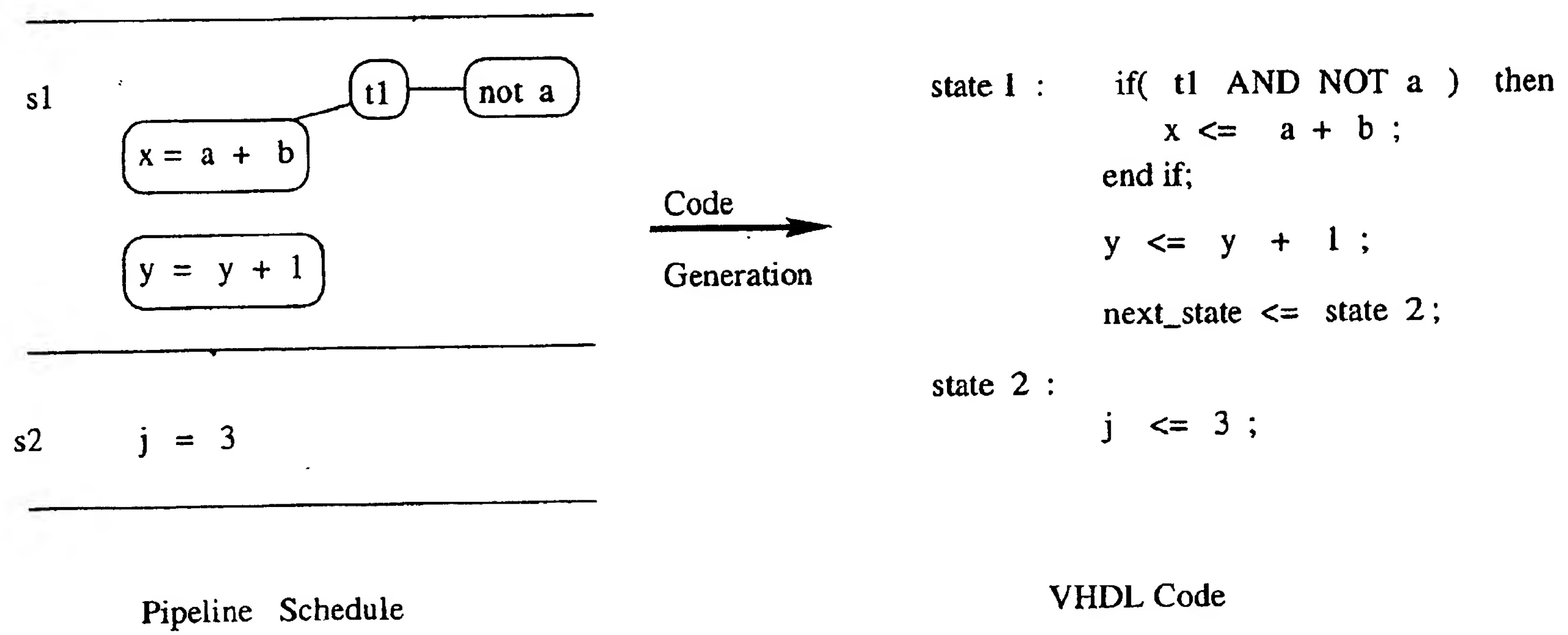
Pipeline Schedule

```
       Copy 0               Copy 1
s1    x = i + 1 ;  x live range         Copy 2
s2    y = i * 2 ;    x = i + 1 ;  x live range
s3    k = x + 3 ;    y = i * 2 ;    x = i + 1 ;  x live range
s4                   k = x + 3 ;    y = i * 2 ;
s5                                  k = x + 3 ;
```

Pipeline Schedule with Overlapping Live Scalars Renamed

```
         Copy 0
s1    case mod_var is
         when 0 :  x0 = i + 1;
         when 1 :  x1 = i + 1,
         when 2 :  x2 = i + 1;
      end case ;
                                           Copy 1
s2    y = i * 2 ;                        case mod_var is
                                            when 0 :  x1 = i + 1;
                                            when 1 :  x2 = i + 1;
                                            when 2    x0 = i + 1,
                                         end case ;                      Copy 2
s3    case mod_var is                                                  case mod_var is
          when 0    k = x0 + 3;          y = i * 2 ;                      when 0 :   x2 = i + 1;
          when 1 :  k = x1 + 3,                                           when 1     x0 = i + 1;
          when 2 :  k = x2 + 3:                                           when 2 .   x1 = i + 1;
      end case ,                                                       end case ;
s4                                       case mod_var is
                                            when 0    k = x1 + 3,      y = i * 2 ;
                                            when 1 .  k = x2 + 3,
                                            when 2 ·  k = x0 + 3,
                                         end case ,
s5                                                                     case mod_var is
                                                                          when 0    k = x2 + 3;
                                                                          when 1 :  k = x0 + 3,
                                                                          when 2    k = x1 + 3;
                                                                       end case ,
```

Figure 19: Renaming of scalars with live overlapping ranges in the pipeline schedule



```
state 1 :    if( t1 AND NOT a ) then
               x <= a + b ;
             end if;
             y <= y + 1 ;
             next_state <= state 2 ;
state 2 :
             j <= 3 ;
```

Figure 20: VHDL Code generation Illustration

METHOD AND APPARATUS FOR AUTOMATICALLY GENERATING HARDWARE FROM ALGORITHMS DESCRIBED IN MATLAB

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0001] This invention was made with Government support by Defense Advanced Research Projects Agency (DARPA) under Contract Number F30602-98-2-0144. The Government may have certain rights in the invention.

FIELD OF INVENTION

[0002] The invention relates to electronic design automation, particularly to the synthesis of hardware from a high-level behavioral description.

BACKGROUND OF INVENTION

[0003] Certain high-level languages, such as MATLAB, are used for prototyping algorithms in domains such as signal and image processing, simulation, analysis, etc. In particular, MATLAB provides users with extensive libraries of high quality routines, as well as high-level matrix-based syntax for expressing computations in a concise manner, i.e., than available from conventional languages, e.g., C, Fortran.

[0004] However, because MATLAB is an interpretive language, programs thereof incur high overhead during runtime. Thus, users developing applications for parallel heterogeneous systems often prototype algorithms in MATLAB, then manually develop algorithms in C, assembly language for DSPs (Digital Signal Processors), embedded processors or in VHDL (VHSIC Hardware Description Language) or Verilog for synthesis and implementation on FPGAs (Field Programmable Gate Arrays) or ASICS (Application Specific Integrated Circuits). Such a manual process is tedious, inefficient, time-consuming, expensive, and unoptimal. Moreover, as hardware designs become faster and include more devices, improved software is needed for hardware synthesis.

SUMMARY OF INVENTION

[0005] The proposed novel electronic design tool and methodology enables automatic synthesis from programming languages, such as MATLAB. A MATLAB program is compiled into a high-level format, such as RTL-VHDL (Register Transfer Level—VLSI Hardware Description Language) or RTL Verilog, which is synthesized using computer-assisted tools to develop ASIC masks or FPGA configurations. Present methodology and system employs an array-oriented programming language such as MATLAB, having a large number of associated functions providing various constructs, such as operation on multi-dimensional arrays, function call statements, conditional statements, or loop statements.

[0006] Additionally, intermediate transformations and optimizations provide optimized RTLVHDL and RTL Verilog description of given MATLAB program. Optimization may include levelization, scalarization, pipelining, type-shape analysis, memory optimizations, precision analysis, or scheduling.

BRIEF DESCRIPTION OF DRAWINGS

[0007] FIG. 1 shows flow chart of preferred method generally according to one aspect of present invention.

[0008] **FIG. 2** shows representative abstract syntax tree according to one aspect of present invention.

[0009] **FIG. 3** shows representative levelization according to one aspect of present invention.

[0010] **FIG. 4** shows representative translation of simple MATLAB program into finite state machine according to one aspect of present invention.

[0011] **FIG. 5** shows representative handling of conditional code for input MATLAB program according to one aspect of present invention.

[0012] **FIG. 6** shows representative handling of loops in input for MATLAB Program according to one aspect of present invention.

[0013] **FIG. 7** shows representative function call in MATLAB and function translation into corresponding state machine according to one aspect of present invention.

[0014] **FIG. 8** shows representative levelization and subsequent translation of array statement of MATLAB program according to one aspect of present invention.

[0015] **FIG. 9** shows representative finite state machine code section according to one aspect of present invention.

[0016] **FIG. 10** shows representative VHDL code generated for finite state machine according to one aspect of present invention. The corresponding Verilog code generation is similar.

[0017] **FIG. 11** shows representative real variables according to one aspect of present invention.

[0018] **FIG. 12** shows representative loop unrolled for memory packing according to one aspect of present invention.

[0019] **FIG. 13** shows representative overall framework for pipelining optimization according to one aspect of present invention.

[0020] **FIG. 14** shows representative terms for pipelining framework according to one aspect of present invention.

[0021] **FIG. 15** shows representative construction of nodes from MATLAB statements according to one aspect of present invention.

[0022] **FIG. 16** shows representative node construction for array access statements according to one aspect of present invention.

[0023] **FIG. 17** shows representative pipeline method according to one aspect of present invention.

[0024] **FIG. 18** shows representative construction of pipeline schedule from loop body schedule according to one aspect of present invention.

[0025] **FIG. 19** shows representative renaming of scalars with live overlapping ranges in pipeline schedule according to one aspect of present invention.

[0026] **FIG. 20** shows representative VHDL code generation according to one aspect of present invention. The corresponding Verilog code generation is similar.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENT

[0027] FIG. 1 shows overview of computer-automated electronic design or compilation process, which may be implemented in one or more software programs and computers or processing elements provided stand-alone or in network or otherwise distributed configuration. According to one operational mode of present automated approach, one or more digital circuit or system may be synthesized or otherwise defined from algorithm described in MATLAB or other programming language. Preferably, MATLAB program is compiled into high-level code, such as Register Transfer Level (RTL)—Very High Speed Integrated Circuit (VHSIC) High Definition Language (VHDL) or RTL Verilog, which is synthesizable using system-specific tools to develop Application Specific Integrated Circuit (ASIC) masks, Field Programmable Gate Array (FPGA) configurations, or other circuit implementations.

[0028] As described further herein, intermediate transformations and optimizations may be performed to obtain highly optimized description in RTLVHDL or RTL Verilog of a given MATLAB program. Additionally, optimizations include levelization, scalarization, pipelining, type-shape analysis, memory optimizations, precision analysis, scheduling, and other operations.

[0029] As shown in FIG. 1, initially one or more input MATLAB program codes 10 are parsed or otherwise processed 12 using one or more directives 36 to build one or more Abstract Syntax Trees (AST) preferably according to the intermediate format (e.g., MATCH Intermediate Format (MIF)). Such an intermediate format may follow standard syntax information based on MATLAB grammar, as well as one or more explicit or implicit indications for design optimizations. Optionally, input code may contain user-specified directives 36 regarding types, shapes, and/or precision of arrays. Directives 36 are attachable to MIF nodes as annotations.

[0030] Then, the type-shape analysis and inference phase 14 is applied. Because, by default, MATLAB variables have no notion of type or shape, type-shape analysis phase 14 analyzes input program to infer type and shape of variables present. Next, the scalarization phase 16 is applied, where operations on matrices may be expanded into loops according to the internal format. When one or more optimized library functions is available for a particular operation one of the library functions is used instead. Further, after the scalarization step 16, levelization 18 may be applied, where one or more complex statements are brokendown into simpler representative statements. Scalarization 16 facilitates VHDL and Verilog code generation and/or optimizations.

[0031] Preferably, the transformation steps 12, 14, 16, 18 are performed on the MIF AST format, and the output of such transformations is also in an MIF AST format. Moreover, hardware-related optimizations may be performed subsequently on such MIF AST files. For example, the precision analysis or inference scheme 20 is applicable to find the minimum number of bits required to represent each variable in the MIF AST based on information available at compile time.

[0032] In addition, the memory optimization or transformation 22 may then be performed on MIF AST for optimization according to memory accesses present in the program as well as the characteristics of the external memory, i.e., when specified as an external input. Furthermore, the pipelining step 24 performs optimizations related to resources present and opportunities of parallel execution and pipelining available. Then, preferably, the MIF AST is translated 26 using RTL-VHDL or RTL Verilog grammar into an RTL-VHDL AST or an RTL Verilog AST 28. Finally, using one or more software intellectual property cores 32, tree traversal 30 of the optimized RTL-VHDL or RTL Verilog AST produces output code in RTL-VHDL or RTL Verilog 34.

[0033] The input MATLAB code is parsed using a formal grammar, and an abstract syntax tree is generated. FIG. 2 shows a graphical view of (a) the hierarchy captured by representative grammar, (b) a sample code snippet, and (c) an abridged syntax tree for code snippet. Parsing and generation of AST are shown, such that MATLAB program is thereby provided using one or more .m" files. Each of such files may include one or more functions, wherein each listed function is defined per one or more statements listed. For example, each statement can contain "if", "while" or "for" statement or a simple expression; each expression may correspond to an "atom", "operator", or a function call. Each atom can be constant or a variable. FIG. 2(*b*) shows a sample MATLAB program, and FIG. 2(*c*) shows the main components of the AST for the sample program.

[0034] Using user-specified directives 36, type-shape information can be provided to the present compilation process; and such directive information may be parsed for annotating the MIF AST. Hence, after the MIF AST is constructed, the compilation process invokes a series of phases, each phase processing the MIF AST, either by modifying or annotating the MIF AST with more information. Directives 36 serve as comments to the compiler, and thus may be used to allow user to provide more information to compiler about program to facilitate optimizations. For example, directive 36 may indicate when design information array includes items whose size at most will be a byte, such that compiler may optimize memory usage accordingly to reduced design space.

[0035] Using MATLAB-type program, type-shape analysis 14 of variables is accomplished effectively by carrying explicit data type and shape information, although MATLAB processing is generally interpretive, whereupon types of variables could be known at runtime before executing a statement. Hence, to compile and synthesize program written in MATLAB, such that maximum information about type and shape of arrays in particular, and of variables in general, are determined appropriately, algebraic framework is thereby provided to determine type and shape of arrays preferably at compile time. Representative directives (e.g,. constraints, assertions, and hints) are provided as follow:

[0036] 1. Constraint directives: State delay and throughput constraints at different levels of granularity. Constraint directives include resource constraint directives that specify resources available and their costs., e.g., %!match DELAY 200 ms suggests to compiler maximum delay of 200 msec to complete an entire application task.

[0037] 2. Assertions: Include assertions made about input MATLAB code, such as variable type assertions, value assertions, etc., e.g., %!match BITS(32) defines a 32-bit variable; helps invoke libraries for FPGAs or synthesize hardware with the right precision, but no more than necessary.

[0038] 3. Hints: Suggestions to compiler, likely to improve performance, including parallelism hints, data distribution hints, platform preference hints, variable type/shape hints etc., e.g., %!match DISTRIBUTE foo(CYCLIC(4), CYCLIC(4)) ONTO PROC(2,2) defines distribution of the variable foo on a 2×2 processor mesh.

[0039] Scalarization **16** is applied to the MATCH intermediate format description for performing source-to-source transformation to a target language. In such step, the target language is typed statically, and only elemental operations are supported.

[0040] In preferred implementation, a high-level programming language is used, such as MATLAB, which is array-based, having built-in functions for supporting array operations. Moreover, to generate therefrom the low-level format, such as VHDL AST or Verilog AST, the corresponding MATLAB MIF AST is scalarized. Thus, to scalarize MATLAB vector constructs, array shape and size are determined; although MATLAB is dynamically typed and may not ordinarily provide explicit basic data type and shape declarations. Accordingly, in accordance with one aspect of present invention, type-shape analysis **14** is applied.

[0041] Generally, translation is provided from one language having array constructs (e.g., MATLAB) to another language having loops and scalar operations (e.g., C), and scalarization may be performed upon intermediate format description (e.g., MIF-AST) to enable translation of array statements into loop form.

[0042] In particular, during operation of present methodology, given certain types and shapes of variables, for example, C-code may be generated to declare variables and corresponding statements. In this regard, compiler software may infer loop bounds for loops corresponding to vector statements provided preferably in MATLAB Following is a sample MATLAB code:

```
a=b+2;
```

[0043] where the correspondingly generated C code is:

```
float a[100; 200]; b[100; 200];
int i; j;
for (i=0; i<100; i++)
for (j=0; j<200; j++){a[i][j]=b[i][j]+2;}
```

[0044] Preferably, the hardware description language, such as VHDL, is used for design file description for simulation and synthesis in accordance with present methodology; although certain constructs, e.g., file operations, assertion statements, or timing constructs may not be supported. Moreover, certain tools may require a specific coding style for generating hardware accurately. Hence, to enhance tool portability, the present methodology provides compiler that generates VHDL code that is compatible with various commercially-available high-level synthesis tools.

[0045] Furthermore, the VHDL AST format may be used, in addition to AST based on MATLAB grammar, to simplify final VHDL code generation, as well as enable hardware-related optimizations, like memory pipelining. Thus, during such optimizations, clock cycles and states may be introduced. Further, to generate VHDL AST, corresponding MATLAB AST is assumed to be scalarized, since MATLAB language is array-based.

[0046] Levelization **18** is applied to scalarized **16** MIF AST, modifying the AST to have statements in the three operand format only. Advantageously, different operators are spread across different states, so that optimal clock frequency is obtained, as shown, for example, in **FIG. 3**. Levelization enables optimizations, such as operator chaining, resulting in a further optimized clock cycle. Since statements having large number of operations are broken down to a series of statements having one operation only, resources may be reused, as these smaller statements can be distributed across different clock cycles.

[0047] Scalarization and levelization steps **16**, **18** transform input MATLAB code, so that such code includes a series of simple statements with constructs. Like conditionals, loops and function calls. **FIGS. 4*a*-*b*** show the transformation of a series of simple MATLAB statements into VHDL statements that are executed sequentially. The corresponding Verilog statements are similar but are not shown. Here, the state machine is synthesized by putting each simple statement in a state, and transitions between the states are arranged so that the states are traversed sequentially, i.e., one after another in order of their appearance in the MATLAB code. This sequencing results in modeling each state to operate in a clock cycle, while movement between the states is decided by the transition signal. **FIG. 10** shows the VHDL code generated for a representative finite state machine. The corresponding Verilog code is similar.

[0048] Next, during the synthesis flow, the compiler synthesizes one or more state machines traversing states for simple statements. For conditionals, a series of states is produced initially corresponding to the 'then' and 'else' body parts. A state is constructed to evaluate the condition, and transitions from the initial state are arranged so that states corresponding to the then-body are traversed when the condition is true, and states corresponding to else-body are traversed when the condition is false; see **FIGS. 5*a*-*b*,** for example.

[0049] Similar to conditional code, loops are handled such that the state machine is constructed for a body of the loop initially. Then, states are synthesized for initializing the index variable, incrementing such index variable, and checking exit condition of the loop. States are attached around the states for loop body, as shown in **FIGS. 6*a*-*b*.**

[0050] Moreover, in the synthesis process, each function call in the MIF AST is mapped to a state machine in the VHDL or Verilog AST; **FIG. 7** shows the state machine representation for a MATLAB code with a function call. Each function is declared as a process, and the arguments of a function are declared as signals. The function arguments are passed by assigning variables at the calling site to signals corresponding to arguments of the function. To assign variables at the caller site, signal names corresponding to arguments of the function and their ordering are know a priori. In this manner, an earlier pass is made generating the symbol table entry corresponding to each function definition, assigning unique names to signals corresponding to the arguments. Each function has an in:start signal and an out:done signal. The execution of the function is started by calling the function by assigning values to the signals corresponding to the arguments and making high the in:start signal for the called function. The calling function waits for the out:done signal of called function to be high, after which the output signal of the called function holds valid values. Hence, advantageously, resources are shared between each function call, and the present approach is applicable to exploit functional parallelism. Preferably, since different processes may run concurrently, multiple processes may not write to shared signals simultaneously.

[0051] Present compiler declares scalars as variables to facilitate movement of operations across states by optimization phases. Variables corresponding to function arguments are declared signals to be visible outside the process corresponding to the function. Other signal declarations include signals corresponding to memory interface.

[0052] Furthermore, the compiler may map arrays to memory; specification of memory access characteristics is provided as an input. The compiler instantiates registers for scalars, e.g., on FPGAs. The levelization phase ensures that each statement has at most one memory access with no other associated operations. The exact mechanism and signals involved in accessing memory is specified in a file read by the compiler, which uses such information to produce states to read/write memory corresponding to each array access that appears in levelized and scalarized MATLAB code; FIGS. 8*a*-*b* shows an example.

[0053] Precision analysis **20** determines the minimum number of bits required to represent a variable. Since number of required bits relates to maximum and minimum value that variable can attain through program run, precision analysis **20** can be performed by value range propagation. Levelization serves to formulate series of transformations applicable on statements to infer the value ranges.

[0054] Moreover, real variables are represented in a way such that operations are accomplished using integer operators; both operands for any operator are integer or real. In particular, to avoid converting induction variables inside loops to be type promoted to real numbers, so-called temporaries are used. Because the MATLAB language is typed dynamically, without ordinarily representing type and shape of variables, data flow graph is used with single assignment property.

[0055] Precision analysis **20** uses an array-based single static assignment (SSA) representation where each array element that is written into more than once is renamed. Advantageously, increase in the value range of an individual array element does not increase the value range of the entire array, so that precision inferencing becomes more accurate. Precision analysis phase **20** ends once value range of all the variables stabilize. Precision information can be derived from target architecture for which VHDL is generated. Value range propagation benefits optimization approaches, such as constant propagation and dead code elimination.

[0056] Preferably, on reconfigurable computing platforms, fixed point representations may be used, since the dynamic range of variables in image and signal processing applications is relatively small. Further, real number representations are scaled down to a value between −1 and +1 so that the number of bits required to represent a real number is related directly to its resolution or number of digits after decimal point.

[0057] **FIG. 11***a* shows a MATLAB code **62** for multiplication of two real numbers. **FIG. 11***b* shows the normal representation code **64** if both numbers are scaled down by the largest integer value of 255 to get the value within −1 and +1; the number of decimal bits needed to represent the transformed number may be as high as 32 bits, i.e., to limit error in calculating the result, resulting in instantiation of a 64-bit integer multiplier. Further, since variables in the input code have to be scaled down by the maximum integer, this approach results in real variables requiring 32 bits leading to a large consumption of processing resources. Thus, in accordance with one aspect of the present invention, real numbers are represented by integer and fractional parts. **FIG. 11***c* shows the resulting transformed code. Transformation results in instantiation of a 13-bit multiplier, with no error in output calculation.

[0058] As described herein, the number of bits required to represent the integral part of a real number can be deduced from the precision analysis algorithm based on value range propagation. Resolution or minimum number of bits required for the fractional part can be inferred after the error analysis phase. Preferably, real variables have the same number of bits for the fractional part; the number of resolution bits for real numbers is inferred when user specifies using directives; user uses output statement, (e.g., printf,) and defines output resolution; or compiler assumes that since the code was to be executed as sequential MATLAB code which has a default resolution of 4, output variables have a resolution of 4, and back propagate such information in error analysis phase to determine resolution of intermediate real variables. Foregoing analysis provides minimum number of bits required to represent fractional part of real numbers, while precision analysis algorithm in previous section provides minimum number of bits required to represent integer part of real number.

[0059] Additionally, optimal packing order (PO) algorithm is provided for each array, where PO is defined by the maximum number of array elements that can be packed in each memory location. The minimum number of bits required by array elements can be inferred from precision analysis **20**. Since most of images read from MATLAB are stored in 2-dimensional arrays, the precision of input images is inferred by parsing input matrices to obtain the maximum value of various array elements. **FIG. 12** shows a loop described in MATLAB. Since memory packing involves unrolling the loop to find more consecutive array element accesses, dependence-analysis phase may be used to determine any loop carried dependencies.

[0060] Preferably, for memory optimization **22**, memory packing is performed on the innermost loop of a deeply nested loop or innermost dimension of array access, and thus, analysis can be done by the greatest common denominator test (GCD). Since memory packing requires consecutive array accesses across loops, array access patterns are determined across loop iterations. Unroll factor, i.e., number of statements unrolled, of each memory access in a loop is defined by the number of array element accesses across loops located in the same physical memory location. To minimize number of memory accesses, the loop is unrolled by the maximum unroll factor.

[0061] Additionally, pipelining **24** optimizes the number of cycles taken by a design to execute input application, as shown in **FIG. 13**. Upon input of a MATLAB loop statement **70**, the given series of nested loops, check **72** is performed on innermost loop body to determine if the pipelining method is applicable. If it is determined that the inner loop body is suitable for pipelining, then the pipelining algorithm is applied **74**. Initially, the inner loop body is located in the AST, then the nodes are constructed corresponding to statements in the loop body. Predicated nodes are constructed for conditional statements present in the loop body. A data flow graph utilizing nodes corresponding to statements of the loop body is constructed **76**. Scheduling algorithm is applied **78** to the data flow graph. The schedule for loop body is used **80** to construct a schedule for the pipeline; scalars with overlapping live ranges in the pipeline schedule are renamed. Loop conditionals are produced **82** and VHDL or Verilog statements are generated **84** from the pipeline schedule.

[0062] Generally, the pipelining **24** step attempts to pipeline innermost loop in sequence of nested loops, according to two conditions: loop under consideration is innermost loop; and no statement in the loop body depends on data defined by a statement in an earlier iteration, but appears after inner loop body. Body of loop statement includes other statements, which may be of three: simple assignment statements, conditional statements, and loop statements, as shown in **FIG. 14**. Traversal of the AST is performed, and each loop statement is checked for nested loops. Simple assignment statements in the loop body are ignored. If a statement in the loop body is a conditional statement, then the body of the conditional statement is recursively traversed to check for the presence of loops.

[0063] If a loop statement is found in the loop body or by recursively traversing conditional statements in the loop body, the loop is judged to be an outer loop, and pipelining is not applied to such loops; else if no loop statement is found in the loop statement body or by recursively traversing conditional statements present in the loop body, then the loop is considered to be an innermost loop. Loops that originate from scalarization of matrix operations are marked to indicate that they do not have dependencies where statement in loop body depends on data defined by statements in earlier iterations, but appears after in loop body. For loops that do not originate from scalarization of matrix operations, GCD test is performed to check for the presence of dependencies.

[0064] Statements of the loop body are traversed one by one, and a node is constructed corresponding to each statement. Nodes are connected by dependency edges to form a dataflow graph. If conditional statements are present in a loop body, then a check is performed on the body of the conditional statement to ensure statements inside the conditional statement body do not modify any conditional variable of the conditional statement. If statements inside the body of a conditional statement modifies any conditional variables, then pipelining **24** is terminated. For statements inside the body of a conditional statement, nodes are created with predicates, e.g., **15**.

[0065] During VHDL code generation corresponding to a particular node, produced VHDL code is guarded effectively by predicate expressions of such a node. For nodes corresponding to statements in the true path of the conditional statement, the predicate expression is the condition variable. For nodes corresponding to statements in the false path of the false conditional statement, the predicate expression is the negation of the condition variable. In case of nested conditional statements, the predicate expressions from higher nesting are concatenated to form the predicate expression of the node. For statements with array accesses, the procedure is slightly different; for array access statements, location of variable is computed first, i.e., for address calculation.

[0066] Then, after address is calculated, the series of signals are assigned specific to the memory interface in use. Given a multi-dimensional array access, a node is generated corresponding to the address calculation in each dimension. Signals assigned for memory access are specified in an external file read by compiler, and nodes are generated corresponding to each state defined in the external file.

[0067] Furthermore, to construct the dataflow graph, an auxiliary control flow graph is constructed initially. In the control flow graph, node "x" is made a predecessor of another node "y", if an execution path exists starting from the first node of the control flow graph that reaches the node "y" with node "x" immediately before in the path. After the control flow graph is constructed, for each node variable that the node defines, and the variables that the node uses are thereby determined. For each variable used by the node, the control flow graph is traversed upward, and all reaching definitions are located. A dependency edge is added from the node using the variable to all nodes with reaching definitions; such operation is applied to all nodes, and nodes along with the dependency edges define the dataflow graph.

[0068] The scheduling process is applied to the data flow graph, and assigns each node in the data flow graph a state number, then the initiation interval for the pipeline is determined. Initiation interval for a pipeline is the number of clock cycles between the initiation of consecutive iterations. Nodes correspond to statements of a loop body with state number assignments, is referred to as the schedule of the loop body. Nodes not dependent on any other nodes are considered initially for scheduling, and assigned state **0**.

[0069] For a given node, once all the nodes that the node is dependent on are scheduled, such node is ready to be scheduled, and such node is assigned the current state number.

[0070] When all the nodes that are ready in a step are assigned, then the state number is incremented to the next value. Exception occurs while assigning a state to a node corresponding to a memory access. If the node corresponding to a memory access is ready, such node is not assigned immediately the current state number. For nodes corresponding to memory accesses, the state number is determined such that if is closest to the current state number, and that the state number modulo the number of memory accesses in loop body is different from all state numbers modulo the number of memory accesses in the loop body corresponding to memory access nodes for which states have been assigned at that point. Initiation rate of pipeline is set to number of memory accesses in loop body. An example of the process in work is shown in FIGS. **14-17** showing representative construction method statements.

[0071] In **FIG. 17**, the sample dependence graph of loop body to be pipelined is shown at left. Dark vertices denote memory references, while light nodes denote non-memory reference vertices; the initiation rate is 2. After placing first memory reference in state **0**, second memory reference cannot be placed in state **4**, although predecessors are assigned; This constraint is because 4 mod **2** is 0, and 0 mod 2 is also 0, which is assigned. So, the second memory reference is pushed to 5.

[0072] After the scheduling process assigns state numbers to all the nodes, pipeline is constructed. Here, L/I copies of the loop bodies are created, where L is length of the loop body schedule, and I is the initiation interval of the pipeline; see **FIG. 18** for the representative pipeline schedule. L is defined by the largest state number assigned to any node, and I is equal to the number of memory accesses in the loop body; index variable corresponding to the loop is var. In ith copy of the loop body, var is replaced by (var+i), then copies of the loop body are concatenated with an interval of I between the successive copies.

[0073] Next, all scalar variables in pipeline schedule are located, and the nodes defining scalars and the nodes using scalars are determined. States between the definition and use of scalars constitute scalar live range. Live range of each variable in each copy of the loop body that comprise the pipeline schedule is determined. Scalars are located for which the live range in one copy overlaps with the live range in another copy of the loop body. A new version is then created for such scalars for each copy of the loop body. Statements that define or use scalars with overlapping live ranges are converted into case statements. For ith case, (i+j)th instance of the scalar variable is used in jth copy of the loop body, for example, as shown in **FIG. 1**. A variable is defined that acts as counter starting with 0 till ceil(L/I−1).

[0074] Moreover, states from **0** to L-I−1 of the pipeline schedule comprises the prologue of the pipeline; states from L-I to L−1 comprise the kernel of the pipeline. The rest of the states are the epilogue of pipeline. Index variable and modulo variable are initialized at beginning of the pipeline kernel. Modulo variable is incremented at the last state of kernel. The index variable is incremented till n−Ceil(L/i)+1, where n is bound of the origin at loop. If index variable is less than n−Ceil(L/i)+1, the state machine loops back to the first statement of the kernel; else the state machine jumps to the first statement of the epilogue.

[0075] Once the pipeline schedule is constructed, VHDL or Verilog code is generated from the schedule and added to the VHDL or Verilog AST. For each node, the basic statement is VHDL. Predicate list of the node is checked, and if predicate expressions exist, then the expressions are ANDed to form a single condition, which guards the execution of the basic statement of the node. All nodes assigned a state are associated in a single state of VHDL AST; see **FIG. 20**, for example. The last statement of the kernel has a conditional statement depending on index variable count that decides the next state. For the rest of the states, the next state is the state that follows immediately.

[0076] Foregoing described embodiments of the invention are provided as illustrations and descriptions. They are not intended to limit the invention to precise form described. In particular, it is contemplated that functional implementation of invention described herein may be implemented equivalently in hardware, software, firmware, and/or other available functional components or building blocks. Other variations and embodiments are possible in light of above teachings, and it is thus intended that the scope of invention not be limited by this Detailed Description, but rather by claims following.

What is claimed is:

1. Computer-automated electronic design methodology comprising the steps of: accessing a first file comprising an algorithm that is described in an array-oriented programming language such as MATLAB; and generating a second file comprising a digital circuit representation that is synthesized automatically from the algorithm, where the representation is Register Transfer level VHDL or Verilog.

2. The methodology of claim 1 wherein:

the second file comprises a design feature for optimization that is described in an intermediate format.

3. The methodology of claim 1 wherein:

the second file is generated using a user directive, effectively allowing a user to guide a compiler by specifying a variable type or shape for optimization.

4. In an electronic design system comprising a compiler for synthesizing a circuit definition from a high-level definition, a process comprising the step of:

determining by a compiler a type or a shape of a variable or an intermediate temporary variable.

5. Electronic design scalarization process comprising the step of:

transforming a first intermediate-format description into a second intermediate-format description by translating an array statement in the first intermediate-format description into a loop in the second intermediate-format description.

6. In an automated design system comprising a processor for optimizing synthesis, a precision-inferencing method comprising the step of:

determining by a processor a maximum bit precision for a variable for effecting a resource optimization.

7. In an electronic design automation system comprising a compiler for processing a file including a real variable, an error-analysis method comprising the step of:

determining a minimum number of bits for representing a real variable.

8. Compact electronic storage technique comprising the steps of:

identifying a plurality of arrays having a bit precision that is less than an available memory width; and

packing more than one array element into a memory location associated with the available memory.

9. In an electronic design automation system, a levelization method comprising the step of:

transforming a first statement in an intermediate-format description into a second statement associated with only one operation.

10. In an electronic design automation system, a memory-access method comprising the step of:

determining a memory access pattern or a memory access constraint for a design synthesis.

11. In an electronic design automation system, a transformation method comprising the step of:

transforming an intermediate format description for synthesizing an optimized memory access.

12. In an electronic design automation system, a method comprising the step of:

determining a number of states in a finite state machine realization associated with a transformed intermediate format description.

13. In an electronic design automation system, a method comprising the step of:

determining a finite state machine realization of one or more levelized statement in an intermediate format description.

14. In an electronic design automation system, a method comprising the step of:

determining a high-level variable or signal assignment for one or more variable in an intermediate format description.

15. In an electronic design automation system, a method comprising the step of:

identifying a pipeline opportunity in an intermediate format description.

16. In an electronic design automation system, a method comprising the step of:

transforming an intermediate format description to enable pipelining in a produced high-level code.

17. In an electronic design automation system, a method comprising the step of:

producing a pipelined high-level code from a transformed intermediate format description.

* * * * *